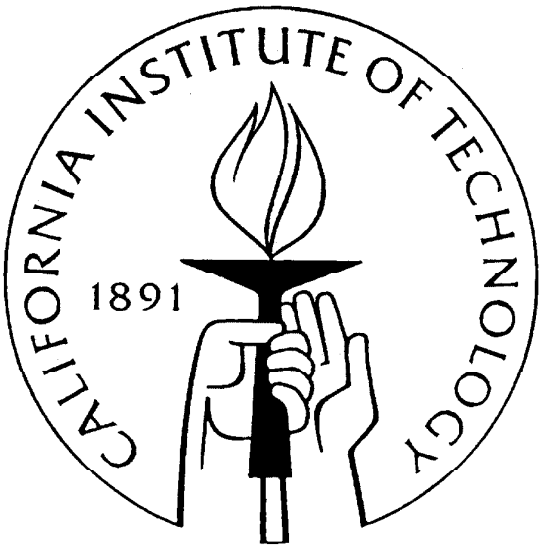


Progress Report to the Advanced Research Projects Agency on the Scalable Concurrent Programming Project

Stephen Taylor

Computer Science Department
California Institute of Technology

Caltech-CS-TR-93-35

Progress Report to the
ADVANCED RESEARCH PROJECTS AGENCY (ARPA)

SCALABLE CONCURRENT PROGRAMMING PROJECT

Department of Computer Science
California Institute of Technology

Technical Category: High-Performance Computing (HPC)
March 22, 1993

Period:	April 27, 1992 to March 22, 1993
Principal Investigator:	Stephen Taylor
Address:	Computer Science 256-80 California Institute of Science Pasadena, CA 91125
Telephone / email:	(818)356-6748 / steve@scp.caltech.edu

1 Project Synopsis

The *Scalable Concurrent Programming Project* is intended to provide significant advances in scalable, high-performance, concurrent programming technology. This is to be achieved through innovations in *compiler technology* and *programming methods*. The research builds on past successes in compiler design, program development tools, programming methods and applications expertise. Two exciting directions provide the central focus for future research: *fine-grain multicomputing* and *irregular applications*. A fine-grain multicomputer is an architecture that reduces the granularity of concurrent computation by integrating hardware support for processes and messages.

The compositional programming systems and techniques that we have developed, PCN [1], *Strand* [3], and FCP [15], have focussed on fundamental concurrent programming concepts. The associated implementation techniques have *simulated*, on medium-grain machines, an underlying fine-grain programming model. The advent of fine-grain multicomputers and their associated "machine language," based on this model, opens two exciting opportunities:

- The design of compiler techniques that compile programs directly into the "machine language" of these architectures.
- The opportunity to rethink the construction of applications and programming tools from the ground up using concurrency.

The "machine language" of fine-grain multicomputers presents a radical change from previous designs and expressing concurrency is a central theme. Previous notions concerning the trade-off between computation and communication have become outdated; new compilation strategies are required to take advantage of this change. The primary activities in which the project is involved are:

1. Programming experiments with existing fine-grain multicomputer software.
2. Construction of program development tools for fine-grain multicomputers.
3. Design of irregular programming techniques.
4. The application of the concurrent programming techniques to non-trivial problems in aeronautical engineering.

Medium-grain multicomputers and existing programming tools are being used as bootstrapping vehicles for fine-grain systems. We are primarily interested in irregular programming problems from a *computer science* perspective: These problems require load-balancing and a substantial volume of communication, thus they provide an excellent basis for pushing the envelope of concurrent programming experiments.

2 Second Year Milestones

The second year project milestones are summarized below:

1. January 1993: Prototype Compiler

2. July 1993: Example Applications (medium-grain)

3 Summary of Progress

During the current funding period, the project research has gained considerable momentum. The project structure has been refined into two groups that work in the areas of *compilation techniques* and *irregular applications*. This year the scope of irregular problems that the group has undertaken has broadened considerably. This has been achieved through collaborative efforts with industrial corporations and other groups at Caltech. During the current funding period the following activities have been completed:

- Four technical papers have been written that describe a variety of analytical, experimental, and applications results. The first concerns an irregular, multibody fluid dynamics solver; it has been submitted to Parallel-CFD 93 [16]. The second concerns load-balancing algorithms; it has been submitted to the Journal of Parallel and Distributed Computing [7]. The third concerns compilation techniques for fine-grain concurrent programs; it has been submitted to Software Practice and Experience [10]. Finally, a technical report concerning irregular concurrent algorithms for virtual reality simulation has been written. We plan to expand and submit this report for publication during the summer months [11].

- A 32-node J-machine, a fine-grain multicomputer, has been installed. A variety of compiler development activities have been conducted using this machine.

- A prototype, fine-grain, message-driven programming system, targeted at the J-machine, has been created. This low-level system includes a compiler, linker, archiver, loader, and micro-kernel. The system uses machine coded floating point arithmetic and allows code and data to be distributed across the entire machine. (Milestone 1)

- A uniprocessor version of the PCN programming system, targeted at the J machine, has been produced. This high-level system is now being modified to make use of specialized hardware provided by the J-machine for fine-grain concurrent programming.

- Two prototype fluid dynamics solvers constructed last year have been under refinement and improvement. The first is an industrial strength TVD/Navier-Stokes code that operates on irregular grids. This code has now been extended to deal

with viscous effects, turbulence modeling, implicit solutions, and multibody geometries. The second solver is being used to understand Wavy-vortex flow; this solver has been extended to utilize multigrid methods. We are currently applying these solvers to non-trivial applications problems. (On schedule for Milestone 2)

- A fine-grain irregular adaptive flow solver based on tetrahedral discretization has been extended to deal with complex boundary conditions and improved numerical methods.

4 Compilation Techniques

Low-level Programming System

Daniel Maskit (2nd Year Ph.D. Student)
Yair Zadik (Masters Degree Student)
Chris Ziomkowski (Staff Programmer)

During the current funding period, the compiler group has completed the construction of a prototype low-level programming system for the J-machine. This is an enabling activity for both the development of large-scale applications and the building of high-level programming tools. The programming system includes:

- A GNU-C compiler that has been retargeted for the J-Machine.
- A communications library that allows the programmer direct access to hardware facilities without incurring copying overheads at either the sending or receiving process.
- A complete set of low-level utilities including assembler, linker, archiver, and loader. These support concurrent program development and provide the ability to distribute program code and data.
- A micro-kernel that provides transparent management of code distribution.
- Machine coded single and double precision arithmetic.

This effort has involved close cooperation with the Concurrent VLSI Architecture group at MIT. The single-precision floating point arithmetic code was provided by MIT and integrated into the compiler, the assembler is a modified version of code originally developed at MIT, and the loader tools are built on top of low-level utilities provided by MIT.

The prototype system directly utilizes J-Machine hardware features such as fine-grained, message-driven processes, hardware synchronization, and on-chip associative memory. The details of this work are discussed in the paper *A Message-Driven Programming System for Fine-Grain Multicomputers* [10]. This paper has been submitted for journal publication to Software Practice and Experience.

Although the current system can be used for application development, it cannot be used for production due to the lack of adequate I/O facilities. Neither disk nor terminal I/O are yet available. As a result, the group is currently engaged in the design of a basic concurrent file system to be used by our applications. Our development cycle has been somewhat painful due to the current polling approach to terminal I/O. We expect this problem to be resolved during the summer.

Over the next year we plan to constantly improve, optimize and extend this basic programming system. It will be used for a varicty of experiments with new compiler techniques and will provide the architects with statistical information on a variety of applications.

High-level Concurrent Programming

Daniel Maskit (2nd Year Ph.D. Student)
Wendy Belluomini (Undergraduate)

Much of our work in high-level concurrent programming has focussed on the use of Program Composition Notation (PCN) [1, 4]. The PCN programming system was originally designed to execute on medium-grain multicomputers; it simulates fine-grain hardware through the use of emulation techniques. During the current funding period we have completed the following tasks:

- A uniprocessor implementation of the PCN programming system for the J-machine.
- Modification of the PCN compiler to generate code that utilizes J-machine hardware features directly.
- The design of message-passing support for a multicomputer implementation of the PCN system.

All of this work builds upon the low-level programming system described earlier. We are currently modifying the PCN micro-kernel to execute the new compiled code correctly and expect to have a working parallel PCN system by mid-summer.

New Work. In building this system we have recognized a completely different view of how to compile high-level programs for fine-grain hardware.

Programming with PCN is, in essence, the application of a few basic programming techniques. These techniques are repeatedly combined in different guises to build complex applications. They allow a wide variety of stream communication protocols, distributed data structures, monitor style atomic operations, and termination detection schemes to be implemented without complicating the language semantic. A complete exposition of the techniques can be found in Chapter 3 of [3].

By knowing the form of these generic programming techniques, it is possible to construct new *communication-oriented compiler optimizations* that capitalize on the structure of the technique in use. For example, all stream communication protocols in PCN are denoted using list operations. An alternative implementation strategy may use the same semantic concept without explicit representation of the list; the list notation simply describes the constraint that message order must be preserved.

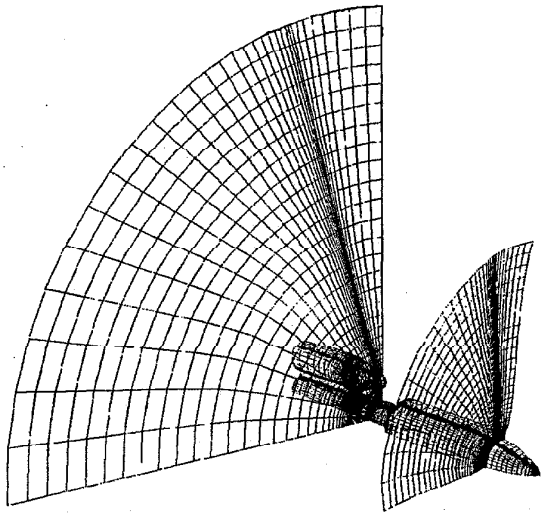
This alternative removes all representation and structure copying overheads from the compiled code. As a result, the efficiency of communication is improved and memory required for message buffering is substantially reduced. Early work in this area has been described in the paper *A Message-Driven Programming System for Fine-Grain Multicomputers* [10]. The thinking behind many of the basic techniques has already been carried out and we plan to experiment with these optimizations during the next year.

5 Irregular Programming Problems

A Concurrent Navier-Stokes Solver for Implicit, Irregular, Multibody Calculations

Johnson C. T. Wang (Fluid Dynamicist, The Aerospace Corp.)

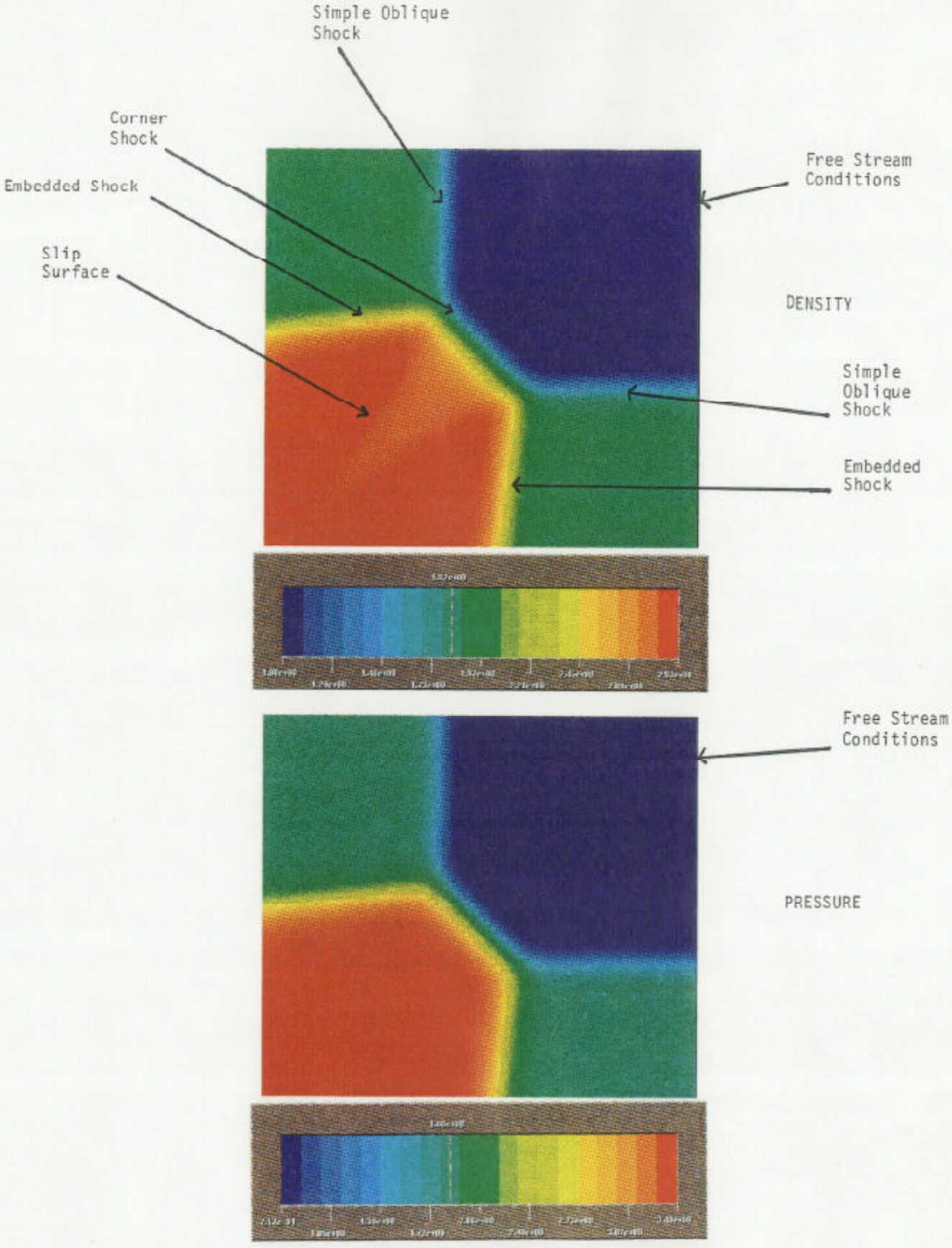
In a collaborative effort with The Aerospace Corporation, the group has recently completed a concurrent implementation of the Aerospace Launch System Implicit/Explicit Navier-Stokes code (ALSINS). This general code is the primary fluid dynamics tool used by The Aerospace Corporation for a broad range of practical simulations. Those of interest involve a variety of nozzle flows and multi-body launch vehicle configurations such as the one illustrated in the figure below. Notice that the computational grid is irregular in structure and the vehicle contains not only a main guidance system but also a booster.



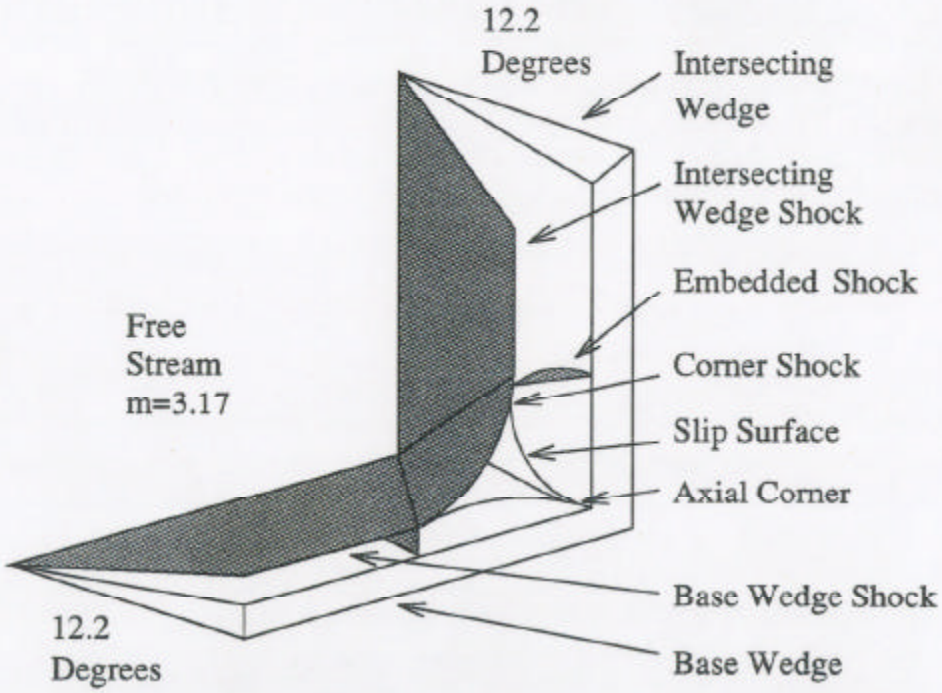
The code utilizes a finite volume TVD scheme for computing both steady state and unsteady solutions to the 3-D compressible Navier-Stokes equations. The scheme is second order accurate in space and is fully vectorized for operation on Cray computers. A line-by-line relaxation algorithm is used to accelerate the convergence for steady state solutions. During the current funding period the group has extended previous inviscid work by adding many new features to increase the practical utility of the code. These features include multibody support, viscous effects, turbulence modeling, and implicit flow capabilities. This work has been described in a paper entitled: *A Concurrent Navier-Stokes Solver for Implicit Multibody Calculations* that has been submitted to Parallel-CFD93.

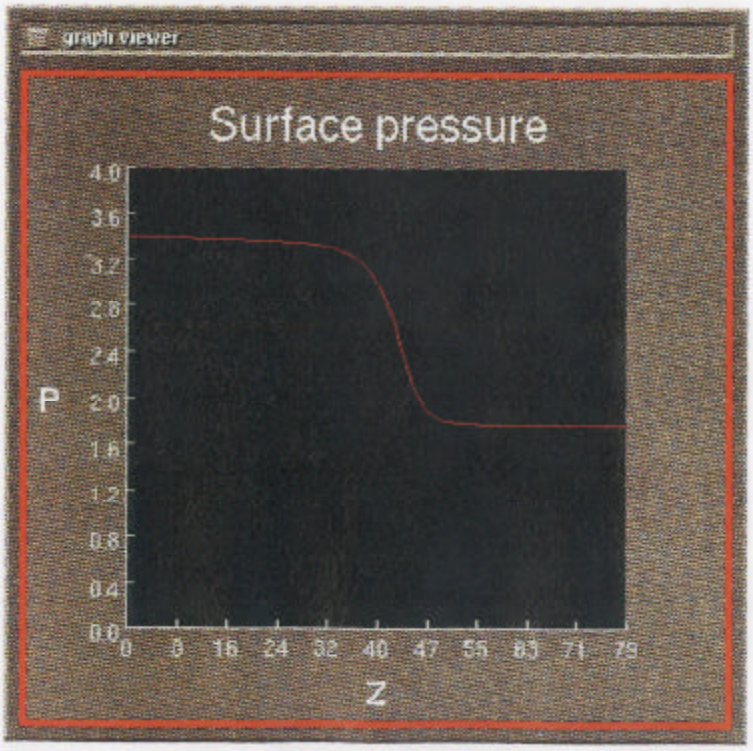
The figure below illustrates experimentally observed steady state effects from a supersonic flow (Mach 3.17) over two inclined faces [2, 14][1]. The color figure on the facing page shows density and pressure plots at the crossection approximately half way through a simulated viscous flow over this geometry. These results were obtained on the Intel Delta

[1]Republished with permission from the Journal of Computational Physics [2]

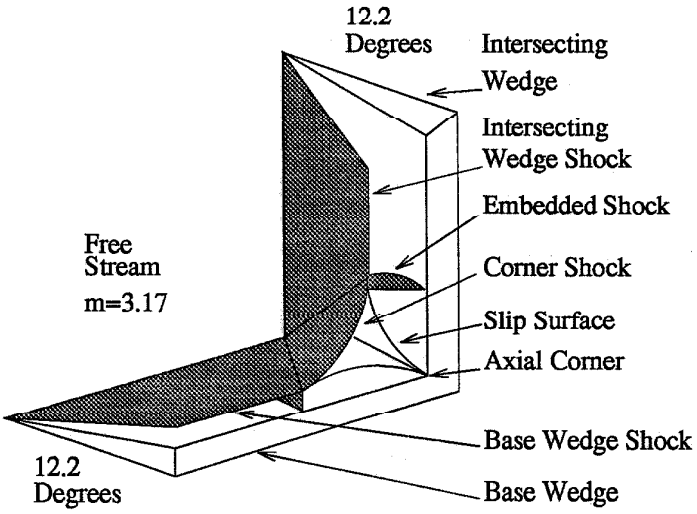

Simple Oblique Shock
Corner Shock
Embedded Shock
Slip Surface
Free Stream Conditions
DENSITY
Simple Oblique Shock
Embedded Shock
Free Stream Conditions
PRESSURE

Numerical Solutions for Supersonic Corner Flow
Shanka, Anderson and Kutler, JCP 17:160-180, 1975
Supersonic Interference Flow Along the Corner of Intersecting Wedges
Charwat and Redekeopp, AIAA 5(3)





machine. The results compare favorably with the experimental data: Notable aspects are the formation of the simple oblique and corner shocks, embedded shocks, and slip surface. The pressure plot shows no pressure change in the area of the slip surface as expected. Inspection of the surface pressure at the wedge shows that the shock is modeled cleanly without ringing.



The ALSINS code is unusual in that it allows complex irregular calculations to be conducted in a manner that is highly vectorizable. We are primarily interested in utilizing the code on *scalable* fine and medium-grain multicomputers for large new simulations. We believe that the high quality of vectorization makes this code a genuine and realistic vehicle for performance comparisons.

We are now applying the code to one of the largest launch vehicle simulations ever conducted: A complete implicit launch vehicle simulation containing 10 individual solid body components. We do not expect the current Intel Delta machine to be capable of conducting this experiment and hope to obtain time on alternative machines by the time we have set up the simulation.

Although the current form of the code is useful, it is far from complete. The main obstacle to optimization and scalability experiments is a result of code being derived from a Fortran code. In the original code all major data structures were placed in common blocks. This unfortunately precludes the storage of more that one block at any computer without substantial rewriting. It also prevents the overlapping of communication and computation that is required to reduce granularity effects and improve load balancing. A second problem results from the static pre-allocation of arrays used in Fortran. This forces the memory allocated to each block to be the size of the *largest* block in the entire domain. This waste of memory results from the lack of dynamic memory allocation; it is a severe restriction if attempting to optimize the use of memory. With current system overheads, these restrictions limit the size of computation we are currently able to execute to approximately five million grid points on the 512 node Delta machine – this needs to be considerably improved.

Load Balancing By Diffusion

Alan Heirich (2nd year PhD student)

This portion of the project is concerned with scalable load balancing techniques for fluid dynamics applications.

Consider the effect of applying multiple heat sources to a block of heat conducting material. The material undergoes temperature changes that do not affect the entire block, but rather, a local region around each heat source. These local regions vary in size according to the nature of the applied heat. After a temperature change the thermal equilibrium of the block is disrupted; however, the cube returns to thermal equilibrium through a process of *heat diffusion.* In the language of differential calculus this process is described by the heat equation

$$u_t = u_{xx} + u_{yy} + u_{zz}$$

Let us apply this analogy to multicomputer programming. Consider the simulation of the three-dimensional flow over some complex geometric shape in which the boundaries move: a space shuttle during separation for example. The computational grid describing the problem must be adapted to follow the movement of boundaries. This is analogous to applying a heat source to a block of material: In a localized region of the flow new computational elements must be added or removed. On a multicomputer, this causes an imbalance in the amount of work allocated to each computer. To rectify this problem we apply a load-balancing algorithm that moves work between computers dynamically.

Our model for load-balancing simulates the heat diffusion process. Thus we are able to derive analytical results by applying standard techniques from numerical analysis. The model is appropriate for quickly diffusing disturbances in a local region of a computational domain. It preserves locality and uses only local communication. Resulting load distributions approximate time asymptotic solutions of the heat equation. As a consequence it is possible to predict both the rate of convergence and the quality of the final load distribution. These predictions suggest that a typical imbalance on a multicomputer with over a million processors can be reduced by one order of magnitude after 105 arithmetic operations at each processor. For large n the time complexity to reduce the expected imbalance is effectively independent of n.

Basic analytical results for this approach to load balancing have been described in the paper *How to Balance a Million Nodes* [7] submitted to the journal of Parallel and Distributed Computing. We are currently performing simulation studies of the algorithm and plan to apply it to non-trivial moving boundary simulations.

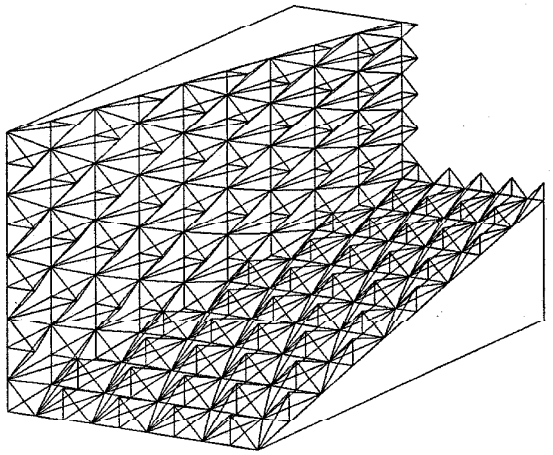
Irregular Tetrahedral Solver

Andy Fyfe (Post-doctoral Student)

This portion of the project is concerned with irregular algorithms for inviscid, compressible flow. The project extends numerical schemes developed by Leveque [8, 9] from two-dimensions to three and develops concurrent algorithms to implement the basic theory. The method is interesting because it allows three-space to be divided into completely irregular regions, requires only local operations, and has a reduced dependence on the size of the time step. We plan to adapt the technique for problems involving moving boundaries.

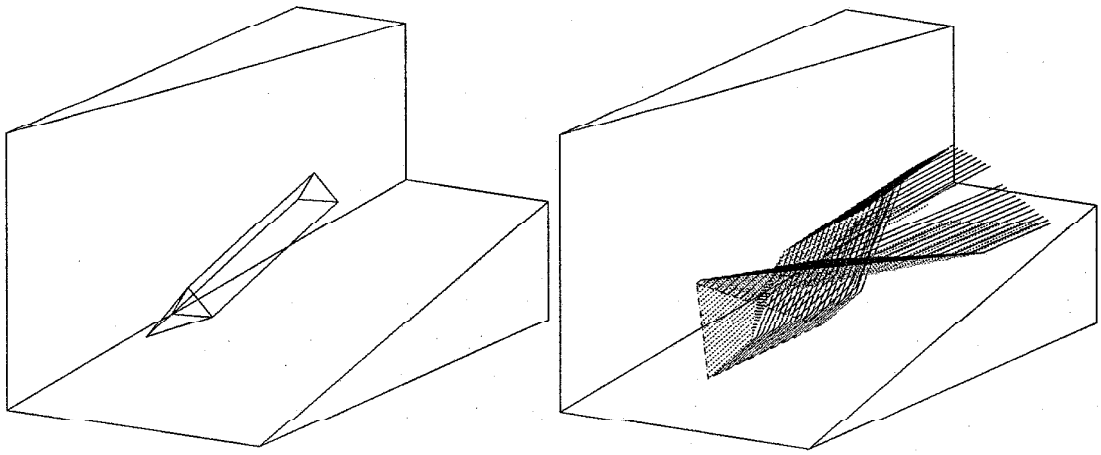
The method is derived from the integral form of the conservation laws for mass, momentum, and energy. Initially, the continuous flow is reduced to a set of dependent variables that represent the average density, momentum and energy within discrete regions of three-space. The continuous flow is approximated by a piecewise constant flow having jump discontinuities at the faces separating adjacent regions. These adjacent regions are then modeled as a *shock tube*. Based on the methods of Roe [13], and Harten and Hyman [6], we approximate the resulting contact surface, shock and expansion waves, allowing them to propagate to neighboring regions for some time Δt.

The concurrent algorithm used to implement this numerical scheme is organized as a collection of communicating fine-grain processes. Each process represents a single tetrahedral region of three-space. Messages are used to pass the necessary wave information from region to region. For concreteness, consider the problem of supersonic flow along a pair of inclined surfaces. The following diagram illustrates this configuration, though only the tetrahedra adjacent to each inclined surface are shown.



The concurrent algorithm requires that processes representing adjacent regions in the underlying geometry exchange their current flow conditions at each time step. Based

on these values, the waves arising from the discontinuities across the common face are approximated. We view these waves as tracing out a prism-shaped volume whose length is the speed of the wave multiplied by the time step and whose base is the common face between the two regions. Such a prism is illustrated on the left in the following figure. For each prism received by a process, the volume of intersection between the tetrahedron represented by the process and the prism is computed. The ratio of these two quantities is used to update dependent variables within the region. Processes track which prisms are received to ensure that they are not affected twice by the same prism. The algorithm also utilizes the time step Δt to ensure that a prism is communicated only to those processes that may possibly be affected by it.



Although open boundaries are simple to characterize, solid boundaries such as those at the corner of the wedge are more difficult. Our approach to this problem is to assume that a wave will reflect off a solid boundary but that where two boundaries come together, the waves are split into separate components. This is illustrated on the right in the figure above.

In our previous work we had used an ad hoc method to compute the intersection between a prism and tetrahedron. We were unable to convince ourselves of its correctness and have subsequently adapted a more general technique to the problem [12]. We are now satisfied with the solution.

Our previous work did not consider solid boundaries and thus could only be used for trivial geometries. We are currently applying the method to the supersonic wedge problem. This problem is an excellent vehicle for validation since it provides a plane of symmetry and experimental data is available. Our other numerical methods have also been validated against this problem; consequently, we are able to compare the quality of the final results.

Immersing the Scientist in Data

Michael E. Palmer (2nd Year Ph.D. Student)

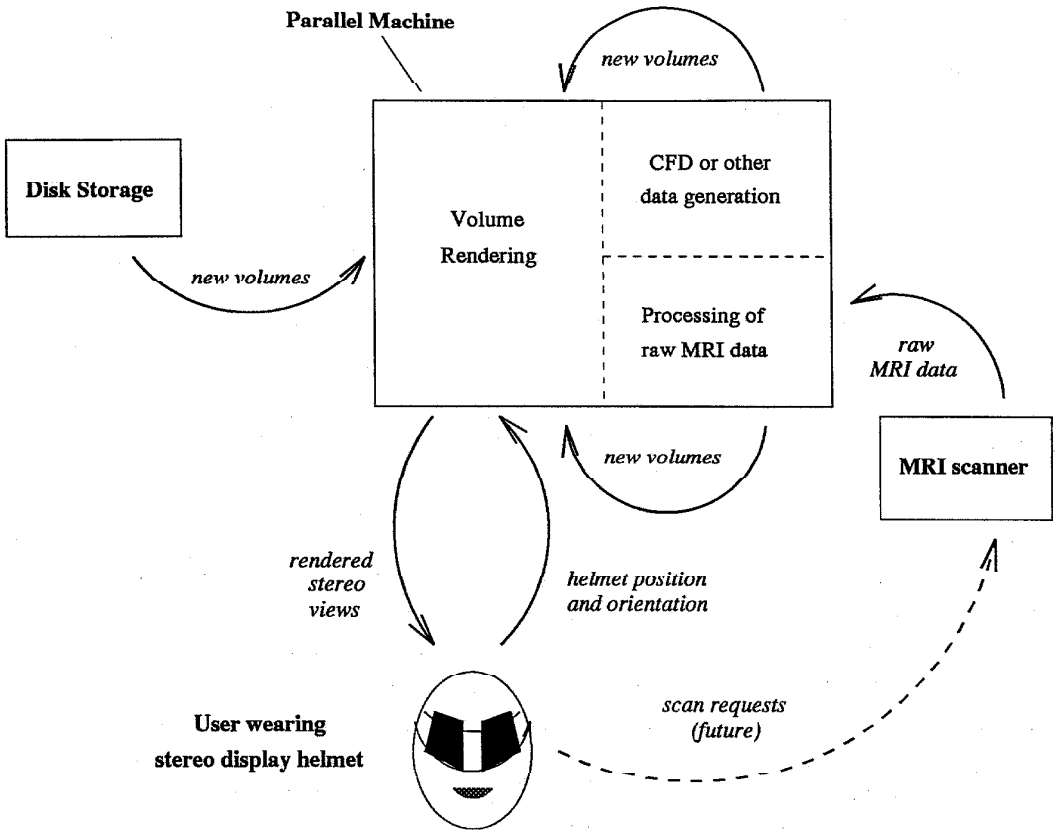
The complex and irregular three-dimensional computations now possible on modern multicomputers are swamping scientists with data; there is too much information to absorb and synthesize into understanding. Currently, scientists scan huge piles of printed output, display two-dimensional cross sections of three-dimensional data, or reduce the volume of information through statistics. These methods may hide interesting details of the data; they fail to exploit the sophisticated apparatus for processing visual information that has evolved in humans. This apparatus allows humans to understand complex three-dimensional, shadowed, colored, and moving environments.

A powerful and intuitive alternative is to immerse the scientist in a running concurrent computation using an interactive virtual environment. Recent advances in multicomputer technology provide the power to simulate virtual environment representations of large computations. The goal of this research is to develop software concepts and strategies now that will be both useful on currently available hardware, and scalable to machines that will be available in the next few years. The visualization tools developed will allow the exploration of a variety of irregular concurrent computations that are under investigation in our laboratory. These include moving boundary problems in Computational Fluid Dynamics (CFD) and real time Magnetic Resonance Imaging (MRI) combining images at multiple resolutions. Our primary concern is to aid the understanding of these computations rather than to render realistic images.

The figure that follows illustrates the proposed hardware configuration in its final form. A concurrent rendering algorithm manipulates three-dimensional scalar data from an MRI scanner, another concurrently executing computation, or snapshots of a computation stored on disk. The scientist wears a helmet that presents a color display to each eye and tracks head movements. The rendering algorithm continually calculates, and displays to each eye, a two-dimensional view of the current volume of data appropriate to the current head position.

The ability to render three-dimensional data interactively at realistic frame rates is within the power of the next generation of massively parallel multicomputers. This advance has been made possible by improvements in communication hardware, such as wormhole routing and the ability to distribute a framebuffer across the end-plane of a multicomputer. Rendering algorithms that are scalable to such multicomputers may be the first to provide interactive rates for direct volume rendering of very large, irregular datasets.

Virtual environment simulations request only a constrained sequence of views due to the continuity of head position and the similarity between views required for the left and right eyes. We have designed concurrent algorithms that exploit these constraints to minimize communication and utilize a straightforward load balancing scheme. The algorithm is structured around fine-grain processes and low-latency communication and is scalable to multicomputers containing many thousands of nodes. Each node is responsible



for rendering a contiguous area of the screen and the volume of screen-space directly behind it. Once the data has been properly distributed, ray casting can be effected entirely within a single node. Further, small changes in viewpoint entail only small shifts in the ownerships of screen-space volume, and therefore require little communication. The computation can be load balanced when necessary by slight shifts in the assignments of screen area to nodes.

This work is described in a Caltech Technical Report *Immersing the Scientist in Data* [11]. Most of the algorithms have already been implemented and we expect this work to mature when distributed frame buffers are available for the J-machine. Recently, this work has been moved from initial ARPA seed funding to an AASERT award provided through the Department of Defense.

Molecular Dynamics

Sharon Brunett (staff programmer)
Kian-Tat Lim (Ph.D. Student, Chemistry and Applied Physics)

This portion of the project is being conducted in collaboration with members of the Department of Chemistry and Applied Physics who are funded as part of an NSF Grand Challenge Award under the direction of Dr. William Goddard. The project concerns a large-scale molecular dynamics application. Early work in this area is based on a particle in cell numerical scheme developed by Kian-Tat Lim. This scheme partitions atoms into cells and represents their interactions in three space using an oct-tree structure. The original application was targeted for the KSR machine, a shared memory architecture.

Initial multicomputer work in this area focused on examining the data structures and algorithms to understand and modularize the code. This process made it possible to abstract the important computational elements into reusable functions. The application was then recoded in terms of these functions and validated on a single node of the Intel Delta machine. During this initial investigation, a significant effort was made to isolate aspects of the algorithms which translate into communication and synchronization on multicomputers.

Due to memory restrictions at each node, it was necessary to rethink major segments of the application. In particular, the initial partitioning of atoms along with their associated oct-tree structure cannot be built and stored in the memory of a single node. The new concurrent algorithm deals with this initial loading and building of the oct-tree by assuming a uniform spatial distribution of atoms. The atoms are associated with computers and the oct-tree structure is then constructed concurrently from the bottom up. The physical problem contains sparse areas which results in very irregular trees and consequently a poor initial load balance.

Algorithms have been designed that systematically balance the load dynamically at intervals as the computation proceeds. Since atoms periodically move between cells and other computers during the computation, load balancing is required irrespective of the initial distribution. Pseudo codes encompassing all of the new algorithms, data structures, communication and synchronization have been completed. To date, the new algorithms for loading an initial set of atoms into the appropriate cells, as well as creating the distributed oct-tree structure have been implemented and validated on a 32 node iPSC860. We are currently involved in integrating the abstract functions that encapsulate the physics of the computation and implementing the load balancing strategy.

We plan to develop the application to production quality and execute it on both the Intel Delta machine and the J-machine. Further, we plan to examine its scalability and use it to examine the structure of composite materials. Dr. Goddard's group continues to improve the physics. These improvements are incorporated into the multicomputer code by maintaining an interface with the abstract code segments we have defined.

Distributed Simulation of Verilog Circuits

Dean Angelico (Sun Microsystems)
Daniel Maskit (2nd year PhD student)
Darren Dang (undergraduate)

This application was undertaken for the specific reason that it does not require substantial floating point arithmetic; consequently, it is the only irregular application being studied by the group that may give insights into fine-grain machines that are not skewed by the absence of floating point hardware.

In conjunction with Sun Microsystems and Chronologic Corporation, we have been developing a switch-level circuit simulation tool that compiles Verilog into C (recall that our low-level programming system is C-based). The tool is specifically oriented toward *regression testing*; a task requiring considerable computing resources at Sun Microsystems. Unlike the general simulation problem, this task provides a variety of useful constraints that can be used in developing concurrent programs.

This project is divided into two parts. The first is concerned with *partitioning* and *granularity control*; the second is concerned with *mapping*.

Initially, a Sun Microsystems tool, Quicknet, is used to restructure a Verilog circuit description using a process of source-to-source transformation. The resulting circuit has the hierarchy of the original design removed and is simply a graph of component subcircuits. This graph is instrumented such that statistics can be gathered on communication and computation patterns. These statistics, along with a model of the parallel architecture, are then used as inputs to a search algorithm which attempts to find a partitioning with minimal communication.

Using the generated partioning, the second part of the project maps the simulation to a multicomputer. This activity utilizes a generic concurrent graph abstraction developed by the research group. The nodes in the graph correspond to subcircuits of the original circuit and are compiled using the Chronologic Verilog compiler. The arcs in the graph correspond to wires and busses between subcircuits. Messages between nodes in the graph correspond to state changes on wires in the circuit. The application employs a novel termination detection scheme to detect the end of each time step in the simulation.

We plan to execute the application on Sun Microsystem shared memory architectures, the Intel Delta machine and J-machine multicomputers. Our hope is to simulate a complete processor design by the end of the year. At this time, the Quicknet tool is not able to handle the entire Verilog language; thus our progress is limited by the speed of extensions to this tool.

Taylor-Vortex Simulation

Alan Heirich (2nd year PhD student)
Andrew Conley (Ph.D. student, Applied Mathematics)

The Taylor-Vortex project was originally undertaken for the purpose of studying scalability and convergence on a non-trivial fluid dynamics problem. In the first stage of this project, reported last year, a concurrent algorithm was developed to perform continuation in solutions of the three dimensional incompressible Navier-Stokes equations [5]. This program used Newton's method to solve an augmented system of equations. Each step of Newton's method was implemented as a parallel red-black relaxation algorithm. The program used finite difference operators supplied to us by the Applied Mathematics department of Caltech.

Subsequent work in this area has yielded an improved algorithm based on multigrid methods. We have attempted to use this algorithm to model a physically observed phenomenon, *spiral vortex flow.* We chose spiral vortex flow because of the availability of experimental data from the American Institute of Physics.

Our attempts to model the phenomenon have not yet succeeded. The concurrent algorithm does not converge for this class of problem. We have learned, the hard way, that red-black algorithms do not propagate boundary information across the computational domain as effectively as alternating direction algorithms. We have also learned that a direct application of the Navier-Stokes equations is not adequate to simulate complex phenomena: In modifying the boundary conditions to permit corotating calculations we uncovered a strange approximation in the finite difference operators. We were forced to remove this approximation in order to implement the new boundary conditions. Unfortunately, removing the approximation brought to light a checkerboard instability which the approximation was evidently meant to suppress.

Despite these setbacks, our interest in this project remains high. We have learned many valuable lessons in attacking a state-of-the-art applied mathematics problem and plan to now step back and try an alternative relaxation scheme. Since we are now conversant with the intricacies of the numerical techniques we intend to immediately target the new code to the J-machine; at this point the entire code has been developed within the group.

6 Industrial Participation

The following projects reported here are being conducted jointly with industrial corporations:

- A Concurrent Navier-Stokes Solver for Implicit, Irregular, Multibody Calculations, with The Aerospace Corporation.
- Distributed Simulation of Verilog Circuits, with Sun Microsystems Corporation.

In addition, members of the group are participating, in an advisory capacity, at the following organizations:

- Northrup Corporation, California; Electromagnetic Scattering and Propagation.
- Philips Labs, California; DSMC Calculations.
- Amoco Production Company, Tulsa, OK; Kirkov Algorithms.

The group has received industrial support from the following corporations:

- Amoco Corporation
- Avalon Corporation
- Intel Scientific Computers
- Northrup Corporation
- Sun Microsystems Corporation
- The Aerospace Corporation

7 Abstracts

How to Balance a Million Nodes

Alan Heirich and Stephen Taylor

We derive analytical results for a dynamic load balancing algorithm modeled by the heat equation $u_t = \nabla^2 u$. The model is appropriate for quickly diffusing disturbances in a local region of a computational domain without affecting other parts of the domain. The algorithm is useful for problems in computational fluid dynamics which involve moving boundaries and adaptive grids implemented on mesh-connected multicomputers. The algorithm preserves task locality and uses only local communication. Resulting load distributions approximate time asymptotic solutions of the heat equation. As a consequence it is possible to predict both the rate of convergence and the quality of the final load distribution. These predictions suggest that a typical imbalance on a multicomputer with over a million processors can be reduced by one order of magnitude after 105 arithmetic operations at each processor. For large n the time complexity to reduce the expected imbalance is effectively independent of n.

A Concurrent Navier-Stokes Solver for Implicit Multibody Calculations

Johnson C. T. Wang and Stephen Taylor

This paper describes a concurrent implementation of the Aerospace Launch System Implicit/Explicit Navier-Stokes code (ALSINS). This general code is the primary tool used by The Aerospace Corporation for complex multi-body launch vehicle configurations and nozzle flow calculations. The code utilizes a finite volume TVD scheme for computing both steady state and unsteady solutions to the 3-D compressible Navier-Stokes equations. The code is second order accurate in space and is fully vectorized for operation on Cray computers. A line-by-line relaxation algorithm is used to accelerate the convergence for steady state solutions.

This paper explains techniques used to generalize the code for operation on scalable multicomputers and describes experiments conducted on the Intel Delta and Paragon systems. The code has been carefully validated using symmetric three-dimensional shock interaction problems for which there is both experimental data and analytical results. The concurrent formulation utilizes domain-decomposition techniques that allow the irregular computational grid to be decomposed into regular blocks of varying sizes. Multibody configurations are modeled through holes in the decomposition. The basic scheme can be load-balanced through standard bin-packing algorithms.

A Message-Driven Programming System for Fine-Grain Multicomputers

Daniel Maskit and Stephen Taylor

This paper describes an experimental message-driven programming system for fine-grain multicomputers. The initial target architecture is the J-machine designed at MIT. This machine combines a unique collection of architectural features that include *fine-grain processes*, *on-chip associative memory*, and *hardware support for process synchronization.* The programming system utilizes these features via a simple message-driven process model that blurs the distinction between processes and messages: messages correspond to processes that are executed elsewhere in the network. This model allows code and data to be distributed across the computers in the machine, and is supported at every stage of the program development cycle. The prototype system we have developed includes a basic set of programming tools to support the model; these include a compiler, linker, archiver, loader and microkernel. Although the concepts are language independent, our prototype system is based on GNU-C.

Immersing the Scientist in Data

Michael E. Palmer, Alan H. Barr, and Stephen Taylor

This paper describes a concurrent algorithm for scientific visualization of irregular, three-dimensional scalar data in an interactive virtual environment. The algorithm is designed to be efficient at direct volume rendering of consecutive frames from nearby viewpoints, such as those arising from smooth head motion or required to display left and right stereo frames.

The main contributions of this work are scalable methods to dynamically partition irregular scalar data for ray casting, to efficiently redistribute this data as the dynamic partitioning changes, and to load balance the rendering computation.

The concurrent algorithm is intended to be scalable to multicomputers containing many thousands of nodes. It is structured around fine-grain parallelism and low-latency communication. Each node is responsible for a contiguous area of the screen and the volume of screen-space directly behind it. The volume allocated to a node changes only slightly with slight changes in viewpoint, and therefore little redistribution of data is usually needed from frame to frame. Furthermore, ray casting can be effected without communication. The computation can be load balanced when necessary by shifts in the assignment of screen area to nodes.

References

[1] Chandy, K. M. and Taylor, S., An Introduction to Parallel Programming, Jones and Bartlett, 1991.

[2] Charwat, A. F. and Redekeopp, L. G., *Supersonic Interference Flow Along the Corner of Intersecting Wedges*, AIAA 5(3).

[3] Foster, I. and Taylor, S., Strand: New Concepts in Parallel Programming, Prentice-Hall, Englewood Cliffs, N.J. 1989.

[4] Foster, I. and Taylor, S., A Compiler Approach to Scalable Concurrent Program Design, California Institute of Technology, Computer Science Technical Report TR 92-07, Submitted for publication, April, 1992.

[5] Harrar II, D., Keller, H., Lin, D., and Taylor, S., Parallel Computation of Taylor-Vortex Flows, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., pp 193–206, 1991.

[6] Harten, A. and Hyman, J. M., Self-Adjusting Grid Methods For One-Dimensional Hyperbolic Conservation Laws. *Journal of Computational Physics*, 50, 235–269, 1983.

[7] Heirich, A. and Taylor, S., How to Balance a Million Nodes, Submitted for publication, March 1993.

[8] LeVeque, R. J., High Resolution Finite Volume Methods On Arbitrary Grids Via Wave Propagation. *Journal of Computational Physics*, 78, 36–63, 1988.

[9] LeVeque, R. J., Computational Fluid Dynamics. Lecture series 1990-03, von Karman Institute for Fluid Dynamics, 1990.

[10] Maskit, D., and Taylor, S., A Message-Driven Programming System for Fine-Grain Multicomputers, Submitted for publication, March 1993.

[11] Palmer, M. E., Barr, A. B., and Taylor, S., Immersing the Scientist in Data, California Institute of Technology, Computer Science Technical Report CS-TR-93-07, March 1993.

[12] Preparata, F. P., and Shamos, M., *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[13] Roe, P. L., Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes. *Journal of Computational Physics*, 43, 357–372, 1981.

[14] Shanka, V., Anderson, D., and Kutler, P., *Numerical Solutions for Supersonic Corner Flow*, JCP 17: 160-180, 1975.

[15] Taylor, S., Parallel Logic Programming Techniques, Prentice Hall, Englewood Cliffs, N.J. 1988.

[16] Wang, J. C. T., and Taylor, S., A Concurrent Navier-Stokes Solver for Implicit Multibody Calculations, Submitted for publication, March 1993.

How to Balance a Million Nodes [1]

Alan Heirich & Stephen Taylor
Scalable Concurrent Programming Laboratory
California Institute of Technology

March 10, 1993

Abstract

We derive analytical results for a dynamic load balancing algorithm modeled by the heat equation $u_t = \nabla^2 u$. The model is appropriate for quickly diffusing disturbances in a local region of a computational domain without affecting other parts of the domain. The algorithm is useful for problems in computational fluid dynamics which involve moving boundaries and adaptive grids implemented on mesh-connected multicomputers. The algorithm preserves task locality and uses only local communication. Resulting load distributions approximate time asymptotic solutions of the heat equation. As a consequence it is possible to predict both the rate of convergence and the quality of the final load distribution. These predictions suggest that a typical imbalance on a multicomputer with over a million processors can be reduced by one order of magnitude after 105 arithmetic operations at each processor. For large n the time complexity to reduce the expected imbalance is effectively independent of n.

[1]The research described in this report is sponsored by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Naval Research under contract number N00014-91-J-1986.

1 Introduction

The emergence of massively parallel computers has introduced new considerations for designers of concurrent algorithms. Two primary issues when dealing with irregular problems are *load balancing* and *scalability.* A variety of algorithms for dynamic load balancing exist for systems with small numbers of computers [2]. These algorithms are elegant, efficient and provably correct. Unfortunately they require that each processor communicate information about it's workload to a central processor which performs the computation. One of our research goals is to develop algorithms which scale to the next generation of fine-grain mesh-connected multicomputers [10, 11] which have potentially hundreds of thousands of processors. Therefore we seek algorithms which emphasize nearest neighbor communication and distribute the computational work across all of the affected processors.

We frequently encounter the need for dynamic load balancing in our work on computational fluid dynamics. This occurs in finite volume or finite difference simulations in which we have partitioned the computational domain across a set of processors on a multicomputer. We can initially distribute the computational grid points in such a way as to make the load on each processor approximately the same. The load remains balanced until the computational grid is adapted to follow shock waves or moving boundaries. These adaptations create grid points in some parts of the domain and destroy points in other parts. As a result the loads on the processors become unequal and dynamic rebalancing is necessary.

This paper describes an efficient algorithm to diffuse local load imbalances on a mesh-connected multicomputer. The algorithm is parameterized and can achieve any desired refinement of balance to the limits of machine precision. It uses no global communication although it does require that processors at exterior points communicate with their counterparts on the opposite sides of the mesh. (We are presently considering strategies to eliminate this requirement.) The algorithm can be used to rebalance a local portion of a computational domain while the normal computation executes concurrently over the rest of the domain. The algorithm can be implemented using policies to minimize the distances over which tasks migrate during their lifetime. This is a desirable property for many simulation problems because it minimizes the cost of communication during the simulation.

We have chosen to model the process of rebalancing by the heat equation $u_t = \nabla^2 u$ on a domain with periodic boundary conditions. Heat diffusion is an apt analogy for the types of problems that we encounter. A load imbalance is typically caused by a computational grid adapting locally and asynchronously. This is analogous to creating a heat source or sink at a point inside a physical volume. Just as heat diffuses away from the source or toward the sink in the physical volume we want work to diffuse away from overloaded processors or toward underutilized processors in the multicomputer. The heat equation is an efficient numerical model for diffusing point sources as it exhibits exponential convergence to equilibrium of all Fourier modes and fast exponential convergence of high wavenumber modes.

Similar informal notions have appeared in the literature [3] such as the Rediflow algorithm of Keller et al [8]. Although such approaches are intuitively persuasive they may also be incorrect and lead to unbalanced loads or unstable behavior. Our algorithm is the

first "diffusive" method for mesh architectures which is scalable and has rigorous proofs of correctness and convergence. It is also the first effort based on a formal model of the rebalancing process and the first demonstration that established numerical methods for grid based computations can lead to practical algorithms for mesh-connected multicomputers. We are currently exploring implementations of this and other models.

The first analytical work on diffusive dynamic load balancing is due to Cybenko [4]. In addition to an optimal diffusive method for hypercube architectures he presents an elegant analysis of a general iterative scheme for arbitrary interconnection patterns. His scheme distributes the computational work across all of the affected processors but does not restrict itself to nearest neighbor communication. Hong, Tan and Chen [6] appreciate the importance of local communication and derive convergence results for nearest neighbor schemes on hypercubes. Unfortunately their results do not apply to mesh-connected architectures and do not take advantage of numerical solution methods for differential equations.

2 The Diffusive Heat Balancing Algorithm

We interpret the workload as a vector $\vec{u}$ which is distributed across the processors of a three dimensional mesh-connected multicomputer of n processors. The algorithm simulates the passage of "artificial time" during which the workload diffuses according to $u_t = \nabla^2 u$. After this time has elapsed the maximum imbalance $|u_{x,y,z} - \bar{u}|$ has been reduced by a given factor α.

1. **Initialization:**

   Choose a desired reduction α in load imbalance. Solve the following inequality numerically for τ (see table I for example solutions)

$$\frac{8}{n}\sum_{i,j,k}\left[1+\alpha 2\left(3-\cos\frac{2\pi i}{n^{1/3}}-\cos\frac{2\pi j}{n^{1/3}}-\cos\frac{2\pi k}{n^{1/3}}\right)\right]^{-\tau} \leq \alpha \tag{1}$$

   where i, j, k are indexed from 0 to $\left(n^{1/3}\right)/2 - 1$ and the case $i = j = k = 0$ is omitted. Determine a parameter ν related to accuracy of the resulting solution.

$$\nu = \left\lceil \frac{\ln(\alpha)}{\ln\left(\frac{6\alpha}{1+6\alpha}\right)} \right\rceil \tag{2}$$

2. **Rebalancing:**

   At every processor x, y, z adjust the workload $u_{x,y,z}$

   `for l=1 to` $\lceil\tau\rceil$ /* outer loop */

   $u_0 = u_{x,y,z}$

for m=1 to ν /* inner loop */

$$u_{x,y,z}^{(m)} = \frac{u_0}{1+6\alpha} + \left(\frac{\alpha}{1+6\alpha}\right)\left(u_{x+1,y,z}^{(m-1)} + u_{x-1,y,z}^{(m-1)} + u_{x,y+1,z}^{(m-1)} + u_{x,y-1,z}^{(m-1)} + u_{x,y,z+1}^{(m-1)} + u_{x,y,z-1}^{(m-1)}\right) \quad (3)$$

endfor /* inner loop */

Exchange $\left(\alpha u_{x,y,z}^{(\nu)} - \alpha u'^{(\nu)}\right)$ units of work with every neighbor u'.

$u_{x,y,z} = u_{x,y,z}^{(\nu)}$

endfor /* outer loop */

$\tau(\alpha, n)$		n (total processors)				
		512	4,096	32K	256K	10^6
α	0.1	8	6	6	5	5
	0.01	297	302	245	214	204
	0.001	6,605	12,589	13,795	11,044	9,895

Table I: Outer loop iterations as a function of multicomputer size n and quality of balance α (see eq. 1).

Discussion. The algorithm consists of two loops. We show in the following sections of this paper that these loops compute an approximate solution to a diffusive process described by the heat equation $u_t = \nabla^2 u$. The outer loop is responsible for advancing artificial time by a step dt. This proceeds until a time has elapsed that is sufficient to diffuse the load imbalance. The inner loop is responsible for computing the new value of the load at a specific time step $t = (l)dt$. When the inner loop has computed the new value an exchange operation causes the load at each processor to equal this value.

Each processor communicates only with it's six neighbors in the three dimensional mesh. A processor which is at an exterior mesh boundary may lack as many as three of these neighbors in the physical mesh. These exterior processors communicate with "logical" neighbors at exterior points on opposite sides of the mesh. This makes the mesh logically spherical and implements the periodic boundary conditions on which our analysis depends.

The cost of the rebalancing phase is the sum of two costs: the cost to compute the new load, represented by the inner loop, and the cost to exchange work among the processors. Each inner loop iteration (3) requires seven arithmetic operations and six bidirectional exchanges of a single number at each processor. The total number of iterations is the product $\tau\nu$. The value of ν is always ≤ 3 for $0 < \alpha < 1$. As table I shows τ is 5 for a rebalancing operation involving one million processors when α is 0.1. Assuming a communication rate within a factor of ten of the instruction rate the cost to reduce the expected load imbalance on a million processor multicomputer by a factor $\alpha = 0.1$ is

about $10\nu(\alpha)\tau(\alpha,\nu(\alpha)) = 10(3)5 = 150$ instruction cycles. This requires less than 4 μs of real time on a multicomputer with 40 MHz processors [11].

For fluid dynamics simulations we would like to preserve task locality while we redistribute portions of the computational domain. This is true for many simulation problems because communication rates are highest between adjacent portions of the domain and as a consequence communication cost is inversely related to locality. In this algorithm locality can be preserved by using an appropriate exchange policy. One such policy would be to assume units of work represent portions of a domain which has been decomposed contiguously across the processors. Under this assumption each neighboring processor represents a neighboring portion of the domain. Any exchange policy which minimizes average pairwise distance between units of work on a common processor maximizes locality.

We would also like to rebalance local portions of a domain without having to interrupt the rest of the computation. The algorithm can be implemented in this way simply by restricting it to a rectangular subportion of the multicomputer mesh. If the algorithm were restricted to a cube then in the following analysis the term n would represent the number of processors in the cube. If the restricted subportion were not a cube then n would be taken as the length of the longest side raised to the third power. This would ensure that convergence bounds reflect worst case estimates.

The reader should understand that the parameter α in the following analysis represents a *reduction* in the expected imbalance rather than a *measure* of the final imbalance. Actual measurements of imbalance require global communication which is contrary to our stated goals. Various implementation policies can be formulated to achieve guaranteed measures of imbalance. For example if the size of the initial disturbance is known then α can be set to remove this disturbance. In this paper we avoid further discussion of the numerous options which exist for implementation policies. Our goal is to establish a sound theoretical basis on which to implement practical algorithms. We are presently considering implementation policies so that we can incorporate these results into our ongoing work in computational fluid dynamics.

3 Derivation Of The Rebalancing Phase

In this section we demonstrate the relationship between our algorithm and the process of diffusion described by the heat equation $u_t = \nabla^2 u$. We first derive a finite difference approximation to the heat equation. This representation is implicit which means that in order to advance the process of diffusion it is necessary to invert a coefficient matrix. We choose to invert the matrix by Jacobi iteration and thus we arrive at the inner loop (3) of the algorithm.

3.1 The Heat Equation And Finite Differences

Consider the parabolic heat equation in three dimensions

$$u_t = \nabla^2 u = u_{xx} + u_{yy} + u_{zz} \tag{4}$$

Taylor expanding in t with all derivatives evaluated at (x,y,z,t)

$$\begin{aligned} u(x,y,z,t+dt) &= u(x,y,z,t) + u_t dt + \mathrm{O}(dt^2) \\ u_t &= \left(\frac{u(x,y,z,t+dt) - u(x,y,z,t)}{dt}\right) + \mathrm{O}(dt) \end{aligned}$$

We obtain the second order terms by expanding in spatial variables where omitted coordinates are interpreted as $(x,y,z,t+dt)$

$$\begin{aligned} u(x+dx,\cdot,\cdot,\cdot) &= u(x,\cdot,\cdot,\cdot) + u_x dx + \\ &\quad u_{xx}\frac{dx^2}{2} + u_{xxx}\frac{dx^3}{6} + \mathrm{O}(dx^4) \\ u(x-dx,\cdot,\cdot,\cdot) &= u(x,\cdot,\cdot,\cdot) - u_x dx + u_{xx}\frac{dx^2}{2} - \\ &\quad u_{xxx}\frac{dx^3}{6} + \mathrm{O}(dx^4) \\ u(x+dx,\cdot,\cdot,\cdot) + u(x-dx,\cdot,\cdot,\cdot) &= 2u(x,\cdot,\cdot,\cdot) + u_{xx}dx^2 + \mathrm{O}(dx^4) \end{aligned}$$

$$u_{xx} = \left(\frac{u(x+dx,\cdot,\cdot,\cdot) + u(x-dx,\cdot,\cdot,\cdot) - 2u(x,\cdot,\cdot,\cdot)}{dx^2}\right) + \mathrm{O}(dx^2)$$

Similar expansions in y,z show that the heat equation can be rewritten

$$\begin{aligned} \frac{u(\cdot,\cdot,\cdot,t+dt) - u(\cdot,\cdot,\cdot,t)}{dt} &= \left(\frac{u(x+dx,\cdot,\cdot,\cdot) + u(x-dx,\cdot,\cdot,\cdot) - 2u(\cdot,\cdot,\cdot,\cdot)}{dx^2}\right) \\ &+ \left(\frac{u(\cdot,y+dy,\cdot,\cdot) + u(\cdot,y-dy,\cdot,\cdot) - 2u(\cdot,\cdot,\cdot,\cdot)}{dy^2}\right) \\ &+ \left(\frac{u(\cdot,\cdot,z+dz,\cdot) + u(\cdot,\cdot,z-dz,\cdot) - 2u(\cdot,\cdot,\cdot,\cdot)}{dz^2}\right) + \mathrm{O}(dt,dx^2,dy^2,dz^2) \end{aligned}$$

Taking $dx = dy = dz$ we can express $u(x,y,z,t)$ in terms of time $t+dt$ and approximate the heat equation to within $\mathrm{O}(dt,dx^2)$

$$\begin{aligned} u(x,y,z,t) \approx \left(1 + 6\tfrac{dt}{dx^2}\right) u(\cdot,\cdot,\cdot,t+dt) - \tfrac{dt}{dx^2}\left[u(x+dx,\cdot,\cdot,\cdot) + u(x-dx,\cdot,\cdot,\cdot)\right. \\ \left. + u(\cdot,y+dy,\cdot,\cdot) + u(\cdot,y-dy,\cdot,\cdot) + u(\cdot,\cdot,z+dz,\cdot) + u(\cdot,\cdot,z-dz,\cdot)\right] \end{aligned}$$

Notice that if we let $\alpha - \frac{dt}{dx^2}$ the preceding equation can be rewritten

$$\begin{aligned} u_{x,y,z}(t) = (1+6\alpha)\, u_{x,y,z}(t+dt) - \alpha\left[u_{x+1,y,z}(t+dt) + u_{x-1,y,z}(t+dt)\right. \\ \left. + u_{x,y+1,z}(t+dt) + u_{x,y-1,z}(t+dt) + u_{x,y,z+1}(t+dt) + u_{x,y,z-1}(t+dt)\right] \end{aligned} \tag{5}$$

3.2 Simulating The Process Of Diffusion

We can consider (5) as a vector equation if we assign coordinates x,y,z to the elements of a vector $\vec{u}$ in the natural way,

$$\vec{u}(t) = A\vec{u}(t+dt) \tag{6}$$

All terms in $\vec{u}$ on the right hand side represent values at time $t + dt$. A finite difference scheme of this sort is said to be "implicit" because values at time t are expressed as a function of values at time $t + dt$. Implicit schemes are known to have desirable stability properties. In particular the error terms do not accumulate in successive solutions so the error in the final solution is $O(\alpha)$. In order to compute solutions at successive time intervals dt we must invert the coefficient matrix A to solve

$$\vec{u}(t + dt) = A^{-1}\vec{u}(t) \tag{7}$$

There are many possible ways to compute $A^{-1}\vec{u}(t)$. In this paper we consider the method of Jacobi iteration [5] as it maintains locality of communication and therefore provides a scalable algorithm. This method determines $\vec{u}(t + dt)$ in solving the system $A\vec{u}(t + dt) = \vec{u}(t)$ with $\vec{u}(t)$ known.

Jacobi iteration splits a coefficient matrix $A = (D - T)$ into a diagonal matrix D and another matrix T with a zero diagonal. With a modicum of algebra we can derive an iteration $\vec{x}^{(m)} = (D^{-1}T)\vec{x}^{(m-1)} + D^{-1}\vec{b}$ from any problem $A\vec{x} = \vec{b}$. This iteration is known to converge to solutions of the original equation if all eigenvalues of $(D^{-1}T)$ lie within the unit circle in the complex plane. A Jacobi iteration is easy to construct from a given coefficient matrix A. D^{-1} is found by inverting each diagonal term d_i independently, while T is just the negation of A with zero diagonal terms. For a finite difference matrix like A in (6) the resulting $(D^{-1}T)$ is just $d^{-1}T$ where $d = d_i$ is the diagonal term which appears in every row of D. Applying this transformation to the vector equation (6) results in the inner iteration (3).

4 Derivation Of Parameters

We have shown that the finite difference representation of the heat equation is accurate to $O(\alpha)$ and we have chosen α to control the quality of the balance which results from simulating the diffusion process. In this section we determine the accuracy of the converged Jacobi iteration and derive a formula for ν which guarantees accuracy α. We then derive a formula for τ which determines the number of "artificial time" steps over which the diffusion occurs.

4.1 Accuracy Of The Jacobi Iteration

From the Geršgorin disc theorem [7] we know the eigenvalues λ of (3) are bounded $|\lambda| \leq \frac{6\alpha}{1+6\alpha}$. Since the row and column sums are constant and the iteration matrix is nonnegative we know further ([7], theorem 8.1.22) that the spectral radius equals the row sum

$$\rho\left(D^{-1}T\right) = \frac{6\alpha}{1 + 6\alpha} \tag{8}$$

Define the error in a current value $\vec{u}^{(m)}$ under the iteration (3) as $e(\vec{u}^{(m)}) = (\vec{u}^{(m)} - \vec{u}^{*})$ where $\vec{u}^{*}$ is the fixed point of the Jacobi iteration. Then for $\nu > 0$

$$e(\vec{u}^{(\nu)}) = e((D^{-1}T)^{\nu}\vec{u}^{(0)}) = (D^{-1}T)^{\nu}e(\vec{u}^{(0)}) \tag{9}$$

which converges when $\rho(D^{-1}T) < 1$ since $\rho((D^{-1}T)^{\nu}) = (\rho(D^{-1}T))^{\nu}$. In order for the algorithm to correctly reduce the error it is necessary to compute the desired load at each time step to an appropriate accuracy. In order to quantify the error define the infinity norm

$$\|e(\vec{u}^{(m)})\|_{\infty} = \max_{x,y,z} \left| e(u^{(m)}_{x,y,z}) \right| = \max_{x,y,z} \left| u^{(m)}_{x,y,z} - u^{*}_{x,y,z} \right|$$

Using this norm we can define a necessary condition for improving the accuracy of the solution $\vec{u}$ by a factor α in ν steps to be $\|e(\vec{u}^{(\nu)})\|_{\infty} \leq \alpha \|e(\vec{u}^{(0)})\|_{\infty}$. From (9) we know that this is satisfied when $(\rho(D^{-1}T))^{\nu} \leq \alpha$ and thus (2)

$$\nu = \left\lceil \frac{\ln(\alpha)}{\ln\left(\rho(D^{-1}T)\right)} \right\rceil \tag{10}$$

4.2 Elapsed Time For The Diffusion

In this section we determine the number of artificial time steps τ required to reduce the load imbalance by a factor α. We do this by considering the eigenstructure of the finite difference equation (5) which we rearrange to express the change in load with each iteration

$$\begin{aligned} u_{x,y,z}(t+dt) - u_{x,y,z}(t) = \alpha \, [& u_{x+1,y,z}(t+dt) + u_{x-1,y,z}(t+dt) \\ & + u_{x,y+1,z}(t+dt) + u_{x,y-1,z}(t+dt) \\ & + u_{x,y,z+1}(t+dt) + u_{x,y,z-1}(t+dt) \\ & - 6u_{x,y,z}(t+dt)] \end{aligned}$$

or as a vector equation with matrix operator L

$$\vec{u}(t+dt) - \vec{u}(t) = \alpha \mathrm{L} \vec{u}(t+dt) \tag{11}$$

Any load distribution $\vec{u}(t)$ can be written as a weighted superposition of eigenvectors $\vec{x}$ of L

$$\vec{u}(t) = \sum_{i,j,k} a_{i,j,k}(t) \vec{x}_{i,j,k}$$

Using this fact we can rewrite the vector equation (11) as

$$\sum_{i,j,k} a_{i,j,k}(t+dt) \vec{x}_{i,j,k} - \sum_{i,j,k} a_{i,j,k}(t) \vec{x}_{i,j,k} = \alpha \sum_{i,j,k} \mathrm{L} a_{i,j,k}(t+dt) \vec{x}_{i,j,k} \tag{12}$$

Using the definition of $\mathrm{L}\vec{x}_{i,j,k}$ and the eigenvalues of L

$$\mathrm{L}\vec{x}_{i,j,k} = -\lambda_{i,j,k} \vec{x}_{i,j,k}$$

$$\lambda_{i,j,k} = 2 \left[3 - \cos\left(2\pi \frac{i}{n^{1/3}}\right) - \cos\left(2\pi \frac{j}{n^{1/3}}\right) - \cos\left(2\pi \frac{k}{n^{1/3}}\right) \right] \tag{13}$$

we can further simplify (12)

$$\sum_{i,j,k} \left(a_{i,j,k}(t+dt) \vec{x}_{i,j,k} \left[1 + \alpha \lambda_{i,j,k} \right] - a_{i,j,k}(t) \vec{x}_{i,j,k} \right) = 0$$

and by the completeness and orthonormality of the eigenvectors

$$a_{i,j,k}(t+dt)\left[1+\alpha\lambda_{i,j,k}\right]-a_{i,j,k}(t)=0$$

$$a_{i,j,k}(dt)=\frac{a_{i,j,k}(0)}{1+\alpha\lambda_{i,j,k}} \tag{14}$$

It is apparent from equation (14) that convergence of the individual eigenmodes is strongly dependent upon the eigenvalues $\lambda_{i,j,k}$. Reducing the amplitude of an arbitrary component i,j,k by α in τ steps of the algorithm requires $[1+\alpha\lambda_{i,j,k}]^{-\tau}\leq\alpha$. The worst case occurs for the smallest positive eigenvalue $\lambda_{0,0,1}=(2-2\cos(2\pi/n^{1/3}))$ which corresponds to a smooth sinusoidal disturbance with period equal to the length of the computational grid. To reduce such a disturbance requires

$$\tau=\left\lceil\frac{\ln\alpha}{\ln\left[1+\alpha\left(2-2\cos\frac{2\pi}{n^{1/3}}\right)\right]}\right\rceil \tag{15}$$

Convergence of this slowest component approaches $\ln\alpha$ for large n since

$$\lim_{n\to\infty}\ln\left[1+\alpha\left(2-2\cos\frac{2\pi}{n^{1/3}}\right)\right]=1$$

Convergence of highest wavenumber component $\lambda_{(n^{1/3})/2-1,(n^{1/3})/2-1,(n^{1/3})/2-1}$ is rapid because

$$\tau=\left\lceil\frac{\ln\alpha}{\ln[1+(6-\epsilon)\alpha]}\right\rceil \tag{16}$$

These characteristics are well suited to our work in adaptive computational fluid dynamics. The disruptions which occur are localized and can be treated as a series of disruptions at individual processors. For these reasons we consider the time to diffuse an expected case in which in a local area of a large mesh becomes imbalanced. We consider the length of time which must pass before the imbalance is reduced by a factor α. Our conclusion is equation (1).

In the following text we use the Poisson bracket $\langle\cdot,\cdot\rangle$ to represent the inner product operator. When discussing loads or eigenvectors we use $\vec{u}[x,y,z]$ or $\vec{x}_{i,j,k}[x,y,z]$ to denote the vector element which corresponds to location x,y,z of the computational grid with the convention that $[0,0,0]$ is the first element of the vector. Then the initial disturbance confined to a particular processor x,y,z can be written as a superposition of eigenvectors of L

$$\dot{u(0)}=\sum_{l,m,n}a_{l,m,n}(0)\dot{x_{l,m,n}} \tag{17}$$

If we assume $u(0)$ to be zero at every element except $[x,y,z]$ then

$$\langle\vec{x}_{i,j,k},\vec{u}(0)\rangle=\vec{x}_{i,j,k}[x,y,z] \tag{18}$$

This is equal to the initial amplitude $a_{i,j,k}(0)$ of each eigenvector $\vec{x}_{i,j,k}$

$$\langle\vec{x}_{i,j,k},\vec{u}(0)\rangle \;=\; \left\langle\vec{x}_{i,j,k},\sum_{l,m,n}a_{l,m,n}(0)\vec{x}_{l,m,n}\right\rangle$$

$$
\begin{aligned}
&= \sum_{l,m,n} \langle \vec{x}_{i,j,k}, \vec{x}_{l,m,n} \rangle \, a_{l,m,n}(0) \\
&= \sum_{l,m,n} a_{l,m,n}(0) \delta_{il} \delta_{jm} \delta_{kn} \\
&= a_{i,j,k}(0) \qquad (19)
\end{aligned}
$$

The computational domain has periodic boundary conditions and as a result the origin of the coordinate system is arbitrary. Then without loss of generality we can place the origin at the source of the disturbance and take $x = y = z = 0$. This has no effect on the eigenvectors $\vec{x}_{i,j,k}$ and from (18), (19)

$$a_{i,j,k}(0) = \vec{x}_{i,j,k}[0,0,0] \qquad (20)$$

Placing the origin at the source of the disturbance is particularly convenient when we consider the first element of the eigenvectors $\vec{x}_{i,j,k}[0,0,0]$. L has $\left(n^{1/3}\right)/2$ distinct eigenvalues $\lambda_{i,j,k}$ each of algebraic multiplicity two. Each of these eigenvalues has geometric multiplicity eight, ie. has eight linearly independent associated eigenvectors of unit length

$$\vec{x}_{i,j,k}[x,y,z] = c_{i,j,k} F_1\left(2\pi \frac{xi}{n^{1/3}}\right) F_2\left(2\pi \frac{yj}{n^{1/3}}\right) F_3\left(2\pi \frac{zk}{n^{1/3}}\right) \qquad (21)$$

where each F_i is either sin or cos. By choosing $x = y = z = 0$ this expression (21) is zero except for the single eigenvector for which $F_1(x) = F_2(x) = F_3(x) = \cos(x)$. As a result without loss of generality we can restrict our consideration to initial disturbances of the form

$$u[0,0,0](0) = \sum_{i,j,k} c_{i,j,k} \vec{x}_{i,j,k}[0,0,0] = \sum_{i,j,k} c_{i,j,k}^2 \qquad (22)$$

¿From (14) we define the time dependent disturbance at any location x', y', z'

$$
\begin{aligned}
\vec{u}[x',y',z'](\tau dt) &= \sum_{i,j,k} c_{i,j,k} \left[1 + \alpha \lambda_{i,j,k}\right]^{-\tau} \vec{x}_{i,j,k}[x',y',z'] \\
&= \sum_{i,j,k} c_{i,j,k}^2 \left[1 + \alpha \lambda_{i,j,k}\right]^{-\tau} \cos\left(2\pi \frac{x'i}{n^{1/3}}\right) \\
&\quad \cos\left(2\pi \frac{y'j}{n^{1/3}}\right) \cos\left(2\pi \frac{z'k}{n^{1/3}}\right) \qquad (23)
\end{aligned}
$$

In the appendix we show that $c_{i,j,k} = (8/n)^{1/2}$ and thus the disturbance is a summation of equally weighted eigenvectors. ¿From (22) and (23) the time dependent disturbance at $0,0,0$ is therefore

$$\vec{u}[0,0,0](\tau dt) = \frac{8}{n} \sum_{i,j,k} \left[1 + \alpha 2 \left(3 - \cos\frac{2\pi i}{n^{1/3}} - \cos\frac{2\pi j}{n^{1/3}} - \cos\frac{2\pi k}{n^{1/3}}\right)\right]^{-\tau} \qquad (24)$$

Solving $\vec{u}[0,0,0](\tau dt) \leq \alpha$ yields equation (1).

5 Algorithm For A Two-Dimensional Mesh

The algorithm for the two dimensional case is very similar to the three dimensional case. The inequality for τ becomes

$$\frac{4}{n}\sum_{i,j}\left[1+\alpha 2\left(2-\cos\frac{2\pi i}{n^{1/2}}-\cos\frac{2\pi j}{n^{1/2}}\right)\right]^{-\tau}\leq\alpha$$

The new spectral radius of the iteration matrix (3) changes ν

$$\nu=\left\lceil\frac{\ln(\alpha)}{\ln\left(\frac{4\alpha}{1+4\alpha}\right)}\right\rceil$$

and similarly the iteration itself

$$u_{x,y}^{(m)}=\left(\frac{\alpha}{1+4\alpha}\right)\left(u_{x+1,y}^{(m-1)}+u_{x-1,y}^{(m-1)}+u_{x,y+1}^{(m-1)}+u_{x,y-1}^{(m-1)}\right)+\frac{u_0}{1+4\alpha}$$

6 Conclusions

We have demonstrated that by starting with a model of the rebalancing problem and applying established numerical methods to that model we arrive at a scalable and concurrent load balancing algorithm for a mesh-connected multicomputer. The essential features of this algorithm are that it diffuses local disturbances rapidly, it can preserve task locality, and it can rebalance local portions of a domain without interrupting the rest of the computation.

We do not claim that this algorithm is optimal and we identify a worst case in which convergence could be very slow for large n. One strategy to remove this worst case behavior would be to embed this algorithm within a multigrid computation [9] in which balances are computed between coarser subdivisions of the architectural mesh. We are continuing the theoretical work on this problem by considering these sorts of alternative numerical methods for the heat equation model as well as alternative models for load balancing. We are continuing the practical work by implementing this algorithm in an adaptive finite volume flow solver for irregular problems with moving boundary conditions. We expect this experience to give us insight into the policy decisions necessary to develop a practical implementation.

7 Acknowledgements

We are indebted to Andrew Conley for insightful discussions and criticisms.

APPENDIX: NORMALIZATION CONSTANT

In this appendix we demonstrate that the eigenvector normalization constant $c_{i,j,k}$ is equal to $(8/n)^{1/2}$ for all eigenvectors $x_{i,j,k}$. From (21) a necessary condition for a normalized eigenvector is

$$\begin{aligned}
1 &= c_{i,j,k}^2 \sum_{x,y,z} \cos^2\left(2\pi\frac{xi}{n^{1/3}}\right)\cos^2\left(2\pi\frac{yj}{n^{1/3}}\right)\cos^2\left(2\pi\frac{zk}{n^{1/3}}\right) \\
&= c_{i,j,k}^2 \frac{1}{8}\sum_{x,y,z}\left(1+\cos 4\pi\frac{xi}{n^{1/3}}\right)\left(1+\cos 4\pi\frac{yj}{n^{1/3}}\right)\left(1+\cos 4\pi\frac{zk}{n^{1/3}}\right) \\
&= c_{i,j,k}^2 \frac{1}{8}\sum_{x,y,z}\left\{\left(1+\cos 4\pi\frac{yj}{n^{1/3}}\right)\left(1+\cos 4\pi\frac{zk}{n^{1/3}}\right)\right. \\
&\quad \left. + \sum_x \cos\left(4\pi\frac{xi}{n^{1/3}}\right)\sum_{y,z}\cos\left(4\pi\frac{yj}{n^{1/3}}\right)\left(1+\cos 4\pi\frac{zk}{n^{1/3}}\right)\right\} \qquad (25)
\end{aligned}$$

We can simplify the preceding expression if we make use of the following

Lemma 1

$$\sum_x \cos\left(4\pi\frac{xi}{n^{1/3}}\right) = 0$$

Proof:

$$\begin{aligned}
\sum_x \cos\left(4\pi\frac{xi}{n^{1/3}}\right) &= \sum_x Rc\left(e^{\imath\frac{4\pi xi}{n^{1/3}}}\right) \\
&= Re\sum_x e^{\imath\frac{4\pi xi}{n^{1/3}}} \\
&= Re\sum_x \left(e^{\imath\frac{4\pi i}{n^{1/3}}}\right)^x \\
&= Re\left[\frac{e^{\imath\frac{4\pi i}{n^{1/3}}}\left(1-\left(e^{\imath\frac{4\pi i}{n^{1/3}}}\right)^{n^{1/3}}\right)}{1-e^{\imath\frac{4\pi i}{n^{1/3}}}}\right] \\
&= 0
\end{aligned}$$

Q.E.D.

Repeated application of lemma (1) to equation (25) yields

$$\begin{aligned}
1 &= c_{i,j,k}^2 \frac{1}{8}\sum_{x,y,z}\left(1+\cos 4\pi\frac{yj}{n^{1/3}}\right)\left(1+\cos 4\pi\frac{zk}{n^{1/3}}\right) \\
&= c_{i,j,k}^2 \frac{1}{8}\left[\sum_{x,y,z}\left(1+\cos 4\pi\frac{zk}{n^{1/3}}\right)+\sum_{x,y,z}\left(\cos 4\pi\frac{jy}{n^{1/3}}\right)\left(1+\cos 4\pi\frac{zk}{n^{1/3}}\right)\right] \\
&= c_{i,j,k}^2 \frac{1}{8}\left[\sum_{x,y,z} 1+\sum_{x,y,z}\left(\cos 4\pi\frac{zk}{n^{1/3}}\right)\right] \\
&= c_{i,j,k}^2 \frac{1}{8} n \qquad (26)
\end{aligned}$$

From which we conclude

$$c_{i,j,k} = \left(\frac{8}{n}\right)^{1/2} \qquad \forall\, i, j, k$$

References

[1] Anderson, D. A., Tannehill, J. C. & Pletcher, R. H. *Computational Fluid Mechanics and Heat Transfer.* Hemisphere, New York, NY, 1984.

[2] Bertsekas, D. P. & Tsitsiklis, J. N. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[3] Chandy, K. M. & Taylor, S. *An Introduction to Parallel Programming.* Jones & Bartlett, Boston, MA, 1992.

[4] Cybenko, G. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.* **7** (1989), 279–301.

[5] Golub, G. H. & Van Loan, C. F. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, 1991.

[6] Hong, J., Tan, X. & Chen, M. ¿From local to global: an analysis of nearest neighbor balancing on hypercube. *Proc. 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.* Association for Computing Machinery, 1988, pp. 73–82.

[7] Horn, R. A. & Johnson, C. R. *Matrix Analysis.* Cambridge University Press, New York, NY, 1991.

[8] Lin, F. C. H. & Keller, R. M. The gradient model load balancing method. *IEEE Trans. Soft. Eng.* **SE-13**, 1 (1987), 32–38.

[9] McCormick, S. F., *Multigrid Methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[10] Noakes, M. & Dally, W. J. System design of the J-machine. In Dally, W. J. (Ed.). *Proceedings of the 6th MIT Conference on Advanced Research in VLSI.* MIT Press, Cambridge, MA, 1990, pp. 179–194.

[11] Seitz, C. L. Mosaic C: an experimental fine-grain multicomputer. *Proc. International Conference Celebrating the 25th Anniversary of INRIA, Paris, France, December 1992,* Springer-Verlag, New York, NY, 1992.

A Concurrent Navier-Stokes Solver for Implicit Multibody Calculations.[1]

Johnson C. T. Wang
Department of Fluid Mechanics
The Aerospace Corporation
and
Stephen Taylor
Department of Computer Science
California Institute of Technology

15th February, 1993

Abstract

This paper describes a concurrent implementation of the Aerospace Launch System Implicit/Explicit Navier-Stokes code (ALSINS). This general code is the primary tool used by The Aerospace Corporation for complex multi-body launch vehicle configurations and nozzle flow calculations. The code utilizes a finite volume TVD scheme for computing both steady state and unsteady solutions to the 3-D compressible Navier-Stokes equations. The code is second order accurate in space and is fully vectorized for operation on Cray computers. A line-by-line relaxation algorithm is used to accelerate the convergence for steady state solutions.

This paper explains techniques used to generalize the code for operation on scalable multicomputers and describes experiments conducted on the Intel Delta and Paragon systems. The code has been carefully validated using symmetric three-dimensional shock interaction problems for which there is both experimental data and analytical results. The concurrent formulation utilizes domain-decomposition techniques that allow the irregular computational grid to be decomposed into regular blocks of varying sizes. Multibody configurations are modeled through holes in the decomposition. The basic scheme can be load-balanced through standard bin-packing algorithms.

[1]This research is sponsored by the Advanced Research Projects Agency, ARPA Order 8176, monitored by the Office of Naval Research under contract N00014-91-J-1986, and The Aerospace Corporation.

1 Introduction

This paper describes a concurrent implementation of the Aerospace Launch System Implicit/Explicit Navier-Stokes code (ALSINS) [7]. This general code is the primary tool used by The Aerospace Corporation for a broad range of practical flow simulations. Those of interest involve a variety of nozzle flows and multi-body launch vehicle configurations such as the one illustrated in Figure 1. Notice that the computational grid is irregular in structure and the vehicle contains multiple bodies: In this instance a main guidance system and booster.

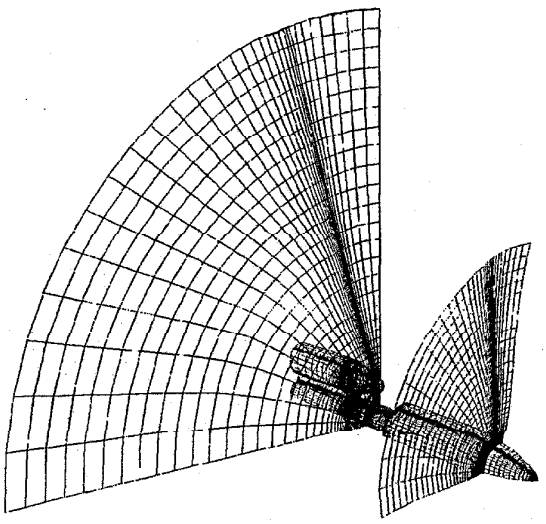


Figure 1: Multibody Launch Vehicle Simulation

The basic governing equations utilized by the ALSINS code are the three dimensional, compressible Navier-Stokes equations in conservation form. The code solves this set of equations in a transformed domain where the irregular computational domain and cells are mapped into a rectangular block with rectangular cells.

The numerical scheme uses a finite volume approach: The governing differential equations are discretized into a set of finite difference equations which equate the change of states at a cell center to the net sum of fluxes across the cell surfaces. This approach allows the three dimensional operator of the Navier-Stokes equations to be reduced to the sum of three one dimensional operations. The inviscid (convective) fluxes are evaluated using an extension of the total-variation-diminishing (TVD) algorithm of Harten [4]. The viscous fluxes are computed using a standard central difference scheme. The numerical solutions are second order in space. The change of states at the cell center are discretized using a first order accurate explicit Euler scheme.

For the steady flow problems, the calculation of the change of states is achieved using an implicit algorithm that allows CFL numbers in the order of hundreds. Using this implicit algorithm, the governing equations are cast into a block tri-diagonal system of

equations with the change of states as dependent variables and net sum of fluxes as non-homogeneous parts. A full set of equations is given in Reference [7]. In this paper we extract only those parts of the equations that are necessary to explain the method of concurrent execution.

The ALSINS code is unusual in that it allows complex irregular calculations to be conducted in a manner that is highly vectorizable. Thus the code executes efficiently on Cray style computers and provides a genuine and realistic vehicle for performance comparisons between parallel machines and vector supercomputers. This paper concerns experiments that utilize the code on medium-grain multicomputers. These machines provide substantial potential due to their *scalability*. We present results for a complex supersonic shock interaction problem executed on the Intel Delta and Paragon systems. This problem has been used to validate all of the concepts described in this paper including inviscid and viscous solutions, implicit techniques, irregular grid decompositions and multiple bodies. The problem provides an interesting tool for validation since it is symmetric and both experimental and analytical data is available.

2 Basic Equations

The numerical scheme is based on the conservative form of the three-dimensional Navier-Stokes equations in the Cartesian coordinate system. These can be written as:

$$\frac{\partial U}{\partial t} + \nabla \cdot \vec{F} = 0 \tag{1}$$

where U is the vector of conserved variables and $\vec{F} = E\hat{e} + F\hat{e} + G\hat{e}$ is the vector form of the fluxes of the conserved variables, given respectively by:

$$U = [\rho\ \rho u\ \rho v\ \rho w\ e]^T \tag{2}$$

$$E = \begin{bmatrix} \rho u \\ \rho u^2 + \rho + \tau_{xx} \\ \rho uv + \tau_{xy} \\ \rho uw + \tau_{xz} \\ (e + p + \tau_{xx})u + \tau_{xy}v + \tau_{xz}w + q_x \end{bmatrix} \tag{3}$$

$$F = \begin{bmatrix} \rho v \\ \rho uv + \tau_{xy} \\ \rho v^2 + p + \tau_{yy} \\ \rho vw + \tau_{yz} \\ \tau_{yx}v + (e + p + \tau_{yy})v + \tau_{yz}w + q_y \end{bmatrix} \tag{4}$$

$$G = \begin{bmatrix} \rho w \\ \rho uw + \tau_{zx} \\ \rho vw + \tau_{zy} \\ \rho w^2 + p + \tau_{zz} \\ \tau_{zx}v + \tau_{zy}v + (e + p + \tau_{zz})w + q_z \end{bmatrix} \tag{5}$$

$$
\begin{aligned}
\tau_{xx} &= -2\mu\frac{\partial u}{\partial x} - \lambda\nabla\cdot\vec{V} \\
\tau_{yy} &= -2\mu\frac{\partial v}{\partial y} - \lambda\nabla\cdot\vec{V} \\
\tau_{zz} &= -2\mu\frac{\partial w}{\partial z} - \lambda\nabla\cdot\vec{V} \\
\tau_{xz} = \tau_{zx} &= -\mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\
\tau_{yx} = \tau_{xy} &= -\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \\
\tau_{yz} = \tau_{yz} &= -\mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial y}\right) \\
\vec{q} &= q_x\hat{e}_x + q_y\hat{e}_y + q_z\hat{e}_z = -K\nabla T
\end{aligned}
\tag{6}
$$

In Equations 1 to 6, ρ is the density, p is the pressure; u, v and w are components of the velocity vector $\vec{V}$ in the, x, y, and z directions respectively with corresponding unit vectors $\hat{e}_x$, $\hat{e}_y$, and $\hat{e}_z$. The total energy per unit volume is denoted by e, q is the heat transfer vector, T is the temperature, λ is the bulk viscosity, μ is the viscosity, and k is the thermal conductivity.

For a polytropic gas, the energy e is related to pressure p by the equation of state:

$$
p = (\gamma - 1)[e - \rho(u^2 + v^2 + w^2)/2] \tag{7}
$$

The viscosity and thermal conductivity are comprised of molecular and turbulent components as $\mu = \mu^M + \mu^T$ and $k = k^M + k^T$, respectively. Here μ^T is evaluated using a Baldwin-Lomax turbulence model[1].

3 Governing Equations in the Transformed Domain

We apply a general coordinate transformation the basic equations of the form:

$$
\xi = \xi(x, y, z); \eta = \eta(x, y, z); \zeta = \zeta(x, y, z) \tag{8}
$$

Thus, Equations 1 can be written in the following form:

$$
\frac{\partial\bar{U}}{\partial t} + \frac{\partial\bar{E}}{\partial\xi} + \frac{\partial\bar{F}}{\partial\eta} + \frac{\partial\bar{G}}{\partial\zeta} = 0 \tag{9}
$$

where

$$
\begin{aligned}
\bar{U} &= U/J \\
\bar{E}J &= E\xi_x + F\xi_y + G\xi_z \\
FJ &= E\eta_x + F\eta_y + G\eta_z \\
\bar{E}J &= E\zeta_x + F\zeta_y + G\zeta_z
\end{aligned}
$$

In Equation 9, J is the Jacobian of the transformation:

$$
J = \frac{\partial(\xi, \eta, \zeta)}{\partial(x, y, z)} = \begin{vmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{vmatrix} \tag{10}
$$

Equation 8 transforms an irregular computational domain such as that shown in Figure 1 into a rectangular one and irregular cells into regular cells. Equation 9 reduces to Equation 1 if $\xi = x$, $\eta = y$, and $\zeta = z$. The beauty of this transformation is that irregular computational domains yield regular domains that can be calculated with a high degree of vectorization on Cray style architectures.

4 Explicit Difference Equations

Carrying out a finite volume integration over the cell and using the first order Euler differencing for the time derivative term, the difference form of Equation 9 becomes:

$$\begin{aligned} U_{i,j,k}^{n+1} &= U_{i,j,k}^{n} - \Delta t \frac{J_{i,j,k}}{\Delta\xi\Delta\eta\Delta\zeta}[\\ &\quad (\bar{E}_{i+1/2,j,k} - \bar{E}_{i-1/2,j,k})\Delta\eta\Delta\zeta + \\ &\quad (\bar{F}_{i,j+1/2,k} - \bar{F}_{i,j-1/2,k})\Delta\xi\Delta\zeta + \\ &\quad (\bar{G}_{i,j,k+1/2} - \bar{G}_{i,j,k-1/2})\Delta\xi\Delta\eta] \end{aligned} \tag{11}$$

where Δt is the time step; subscripts i, j, and k indicate the spatial location of the cell and superscript n the time step. The terms inside the bracket, representing the fluxes across the cell surfaces, are depicted in Figure 2.

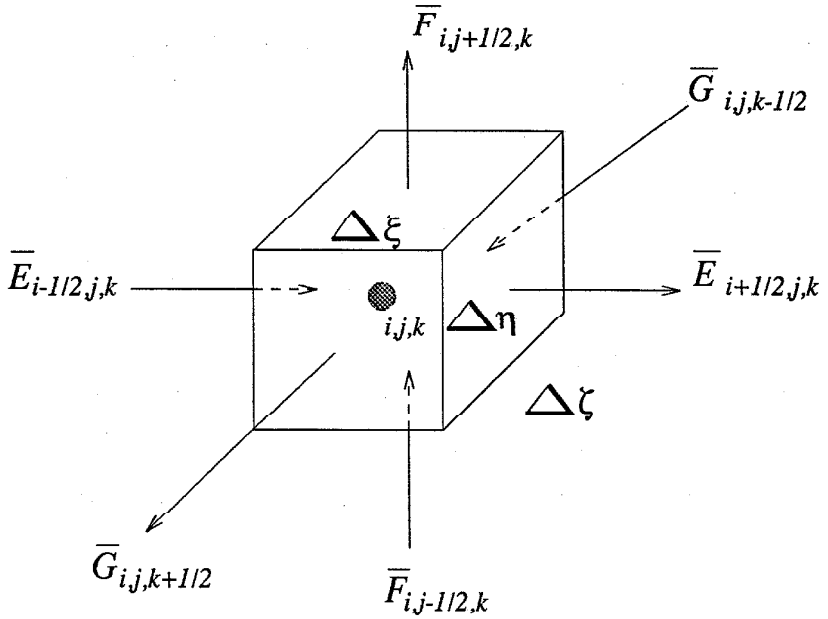


Figure 2: Flux Terms for Finite Volume Calculation

It can be seen from Equation 11 that there are two parts, inviscid and viscous, in each surface flux term. We will use superscript *inv* to indicate the inviscid part and *vis* the

viscous part. For example,

$$\bar{E}_{i+1/2,j,k} = \bar{E}^{inv}_{i+1/2,j,k} + \bar{E}^{vis}_{i+1/2,j,k} \tag{12}$$

Details for computing the surface fluxes are given in References [6, 8, 9, 10]. For the purpose of this paper, it is sufficient to point out that the computation of the inviscid part of $\bar{E}$ requires information concerning $\bar{U}^n$ at positions $i-1$, i, $i+1$, and $i+2$ for second order spatial accuracy, i.e.

$$\bar{E}^{inv}_{i+1/2,j,k} = \bar{E}^{inv}(\bar{U}^n_{i-1,j,k}, \bar{U}^n_{i,j,k}, \bar{U}^n_{i+1,j,k}, \bar{U}^n_{i+2,j,k}) \tag{13}$$

and

$$\bar{E}^{inv}_{i-1/2,j,k} = \bar{E}^{inv}(\bar{U}^n_{i-2,j,k}, \bar{U}^n_{i-1,j,k}, \bar{U}^n_{i,j,k}, \bar{U}^n_{i+1,j,k}) \tag{14}$$

It should be noted that in Equations 13 and 14, the values of j and k are kept the same. The central difference scheme is also used to compute the viscous terms, therefore:

$$\bar{E}^{vis}_{i+1/2,j,k} = \bar{E}^{vis}(\bar{U}^n_{i+1,j\pm1,k}, \bar{U}^n_{i+1,j,k\pm1}, \bar{U}^n_{i,j\pm1,k}, \bar{U}^n_{i,j,k\pm1}) \tag{15}$$

Unlike Equation 13, for a given j and k, $\bar{E}^{vis}$ depends on $\bar{U}^n$ at $j \pm 1$ and $k \pm 1$. This is due to the cross derivatives for the viscous terms; For example, see the definition of τ_{xx} in Equation 6.

5 Implicit Difference Equations

To accelerate the convergence for steady state problems, a numerical technique was presented in Reference [7] using a line relaxation procedure. The discretized equation in the $i-th$ direction is a block tri-diagonal system:

$$\tilde{A}^- \Delta U^n_{i-1,j,k} + \tilde{D} \Delta U^n_{i,j,k} + \tilde{A}^+ \Delta U^n_{i+1,j,k} = RHS \tag{16}$$

where $\tilde{A}^-$, $\tilde{D}$, and $\tilde{A}^+$ are preconditioned 5x5 matrices. The unknowns ΔU^n are defined as:

$$\Delta U^n_{i,j,k} = U^{n+1}_{i,j,k} - U^n_{i,j,k} \tag{17}$$

In Equation 16, the RHS is the second term of the right hand side in Equation 11. For a steady state solution $\Delta U^n = 0$ by definition. When this condition is reached, from Equation 16, it follows that $RHS = 0$. This represents a solution to the steady state Navier-Stokes equations.

Figure 2 shows a computational cell of size $\Delta\xi\Delta\eta\Delta\zeta$ with cell center at ξ, η, ζ and the surface fluxes.

6 Domain Decomposition

The basic numerical scheme outlined in previous sections is implemented on parallel machines using the technique of *domain decomposition.* This technique involves dividing the data structures of the program and operating on the parts independently.

To cope with complex geometries it is necessary to decompose the irregular grids into blocks of varying sizes. However, a completely arbitrary decomposition is not needed. For a wide variety of practical launch vehicle configurations it is sufficient to decompose the axes. Figure 3 shows an example of this type of decomposition. An irregular grid, such as that shown in Figure 1, is first transformed into a regular computational grid through the transformation described in Section 3. The computational grid is then decomposed into blocks of varying sizes along each axis. For example, in Figure 3, decomposition along the X axis yields blocks with X dimensions containing 3, 5, and 4 cells; decomposition of the Y and Z axes yield blocks with dimensions containing 2, 4, and 3 cells.

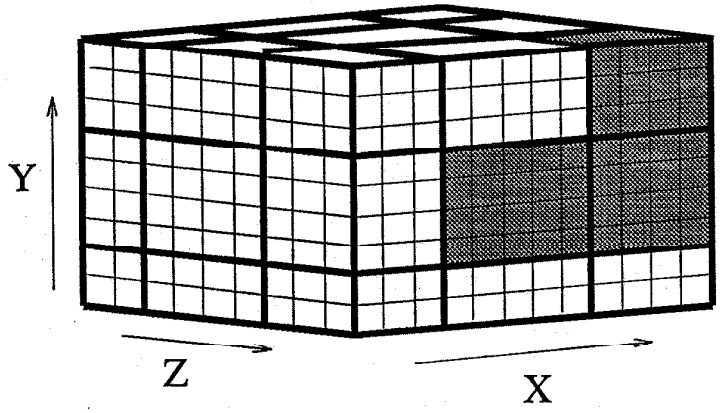


Figure 3: Irregular Domain Decomposition

In generating the decomposition it is necessary to specify the boundary conditions for each block independently. In addition to the standard solid wall and free-stream boundary conditions, we add a further condition that specifies an edge resulting from a cut through the domain.

This decomposition is capable of modeling a variety of single body geometries but is not sufficiently flexible to model multibody launch vehicles such as those illustrated in Figure 1. To generalize the basic decomposition strategy we consider some blocks in the decomposition as *holes*. These holes are surrounded by solid boundaries and thus represent multibody configurations. Figure 3 uses shaded areas to signify the location of holes, representing multiple bodies, in the decomposition.

7 Concurrent Algorithm

Given the decomposition outlined in Section 6 it is straightforward to build an iterative concurrent algorithm. Recall Equations 13 and 14 from Section 4:

$$\bar{E}^{inv}_{i+1/2,j,k} = \bar{E}^{inv}(\bar{U}^n_{i-1,j,k}, \bar{U}^n_{i,j,k}, \bar{U}^n_{i+1,j,k}, \bar{U}^n_{i+2,j,k})$$

and

$$\bar{E}^{inv}_{i-1/2,j,k} = \bar{E}^{inv}(\bar{U}^n_{i-2,j,k}, \bar{U}^n_{i-1,j,k}, \bar{U}^n_{i,j,k}, \bar{U}^n_{i+1,j,k})$$

These equations are used to compute the inviscid flux across a cell surface. The second order accurate numerical scheme requires two cells of information from other blocks at time $t-1$. Thus to compute an entire block in the decomposition, it is necessary to communicate neighboring faces from adjacent blocks in the decomposition.

Viscous Effects. Unfortunately, this communication scheme is not sufficient to calculate the viscous effects as indicated in Equation 15:

$$\bar{E}^{vis}_{i+1/2,j,k} = \bar{E}^{vis}(\bar{U}^n_{i+1,j\pm1,k}, \bar{U}^n_{i+1,j,k\pm1}, \bar{U}^n_{i,j\pm1,k}, \bar{U}^n_{i,j,k\pm1})$$

Recall that this is due to the cross derivative terms of the viscous sheer. Thus each block of the decomposition needs to have available the twelve neighboring corners from adjacent blocks. The shaded regions in Figure 4 illustrate the types of communication structure necessary to compute all cells in a single block at time t based on values at time $t-1$.

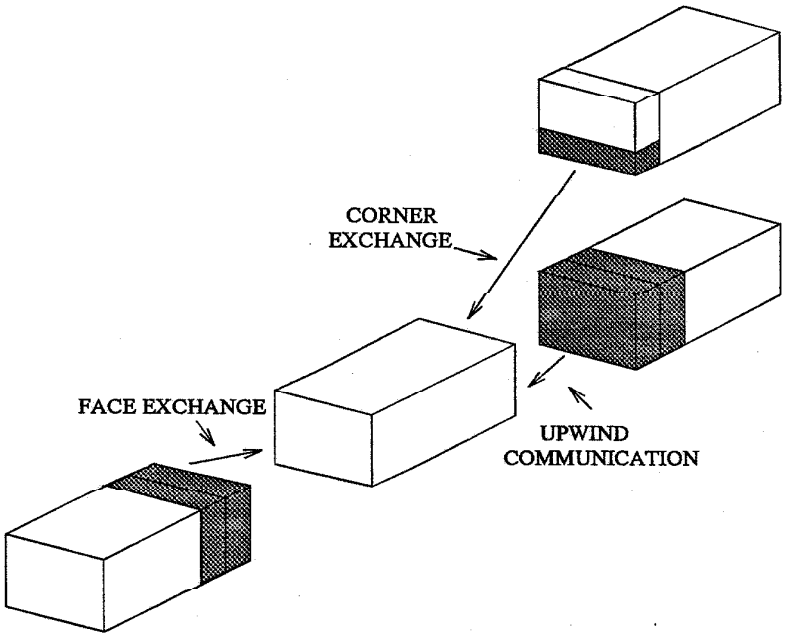


Figure 4: Communication Structure

Implicit Solutions. Although this basic communication structure is sufficient for the explicit simulations, it requires some basic changes to the numerical scheme to handle implicit calculations. Recall, from Section 2, that a line-by-line relaxation scheme is used for this calculation. Unfortunately, this scheme is a sequential algorithm that iterates through an entire dimension of the domain: It cannot, therefore, be executed concurrently within each block. However, in the steady state, the change in dependent variables from one time step to another will be zero:

$$\Delta U^n_{i,j,k} = U^{n+1}_{i,j,k} - U^n_{i,j,k} = 0$$

We take advantage of this fact by setting the required ΔU^n from an adjacent block to be zero. From previous experience in using the sequential code, we know that this approach does not cause instability and even speeds up convergence.

Model for Time. Recall Equation 11 from Section 4:

$$\begin{aligned} U^{n+1}_{i,j,k} &= U^n_{i,j,k} - \Delta t \frac{J_{i,j,k}}{\Delta\xi\Delta\eta\Delta\zeta}[\\ &\quad (\bar{E}_{i+1/2,j,k} - \bar{E}_{i-1/2,j,k})\Delta\eta\Delta\zeta + \\ &\quad (\bar{F}_{i,j+1/2,k} - \bar{F}_{i,j-1/2,k})\Delta\xi\Delta\zeta + \\ &\quad (\bar{G}_{i,j,k+1/2} - \bar{G}_{i,j,k-1/2})\Delta\xi\Delta\eta] \end{aligned}$$

In this equation, the value of Δt is bounded by the Courant number. In order to find the next Δt at step t it is necessary to combine information from all blocks in the decomposition. Although, for steady state problems, the use of a local Δt has been suggested in the literature, our experience is that this yields results which compare poorly with experimental data. Therefore, we use a uniform Δt even for implicit calculations. This is calculated by finding a local minimum Δt for all blocks independently and then combining these values to provide the minimum value over the entire domain.

Turbulence Modeling. To complete the basic algorithm we require a method for utilizing turbulence models on parallel machines. For simplicity we achieve this by assuming that turbulent effects are localized with a single block adjacent to a boundary. This allows all turbulent effects to be calculated without communication or adding sequential operations to the concurrent algorithm.

Single-Body Algorithm. To simplify the implementation, we represent each block in the domain as a *concurrent process* and associate with it communication channels to eighteen neighbors: These channels connect a block to the six neighbors required for face exchanges and twelve neighbors used for corner exchanges. End around connections are used at the boundaries to ensure that every block has a full complement of neighbors. It is trivial to establish the appropriate channels using the techniques described in [2]. In outline the basic algorithm may now be expressed as shown abstractly in Program 1.

A block process can be executed at any computer in the machine. Each block loads the appropriate geometry information from a unique file concurrently. In this manner a block is informed of its boundary conditions. Notice that the algorithm is self-synchronizing by virtue of the explicit number of face and corner exchanges required. In the cases where the boundary conditions signify values communicated are unimportant we send only a single number and discard it at the receiver.

Notice that the load and compute steps in the algorithm are simply existing Fortran code from the original ALSINS code. This code is modified to deal with a single additional boundary condition: that of an edge in the domain.

Multi-Body Algorithm. The scheme is adapted to deal with multibody simulations by allowing any block to represent a hole in the domain. A block is informed that it represents a hole when the geometry information is loaded. If a block represents a hole the **compute** function does nothing, the **extract** functions returns a vector of length 1, and the **calculate** function returns *infinity*.

```
block(...)
double U[...];
{  load geometry data into block
   calculate local Δt
   send local Δt to minimum calculation
   while(time not exhausted) {
      recieve global Δt
      extract 6 faces from block
      send faces to adjacent neighbors
      extract 12 corners from block
      send to diagonal neighbors
      recieve 6 faces and 12 corners
      compute dependent variables at t + Δt using faces, corners, and Δt
      calculate new local Δt
      send new local Δt to minimum calculation
   }
}

minimum(...)
{  recieve a local Δt from each block
   calculate the minimum
   broadcast minimum to all blocks
}
```

Program 1: Basic Iterative Algorithm

Numerous optimizations can be carried out to the algorithm to improve memory use and reduce communication. We do not in general consider the additional complexity of special cases in the communication structure as worthwhile. To understand why, we consider the consider the algorithm analytically and form an expression for the computation and communication at any time step t. Assume that all blocks have uniform dimension n. Then the time to execute a single iteration can be approximated by:

$$time = p(An^3 + 6Bn^2 + 12Cn + Dn^3 + 2E + F) \quad (18)$$

Where p is the number of computers, and A, B, C, D, E, and F are constants. The term An^3 arises from the need to compute n^3 cells. The terms $6Bn^2$ and $12Cn$ arise from the communication of faces and corners respectively. The term Dn^3 represents the computation of a local minimum Δt value for a single block. The value $2E$ represents the communication of a local Δt and global Δt between a block and the location where the global minimum is calculated. F represents the cost of calculating the global minimum Δt.

From Equation 18 we see that the granularity of the computation, or ratio of computation to communication is:

$$(A + D)n^3 + F/6Bn^2 + 12Cn + 2E \quad (19)$$

Thus by picking a suitably sized block any amount of communication can be reduced to insignificance.

8 Load Balancing

Notice that the computation will be reasonably well balanced if the blocks are similar in size. For the most part our blocks are sufficiently large that a simple mapping technique is sufficient. Each block is numbered to uniquely identify it and mapping is achieved by mapping the ith block to the ith computer; thus for the most part placing contiguous blocks close together. Most of our current experiments have been based on this simple technique.

We are currently adapting the code to utilize an alternative bin-packing technique. This technique is appropriate since the grid is not adaptive and computation time is related directly to the number of grid points in a block. The central idea is to pack the blocks into computers by placing the next block at the computer with the *smallest* number of stored grid points. To achieve this organization it is necessary to keep track of how many grid points are allocated to each computer. The goal is to keep the amount of grid points at each computer balanced and thus balance the computation. Figure 5 shows an example bin packing. The number of grid points in a block is represented by the height of a box; blocks are labeled by their identifier and are placed into computers in numerical order.

This scheme has a number of advantages: It optimizes the use of memory and permits a more even distribution of load in the presence of large disparities in the size of the blocks. Processes representing holes and performing global operations can be packed into

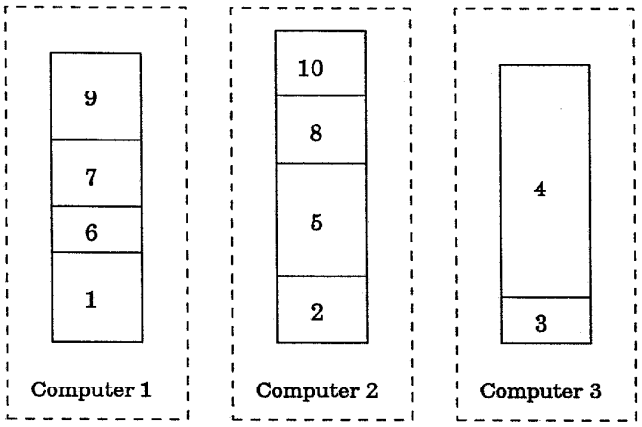


Figure 5: Load Balancing by Bin Packing

a single underutilized computer. Finally, by multiprocessing multiple blocks in a single computer, it is possible to *overlap* the communication associated with one block with the computation of another. This is a basic concurrent programming concept used to overcome latency that has been largely ignored in applications.

The main obstacle to utilizing this bin-packing technique is that the original code was constructed with all major data structures in Fortran common blocks. This unfortunately precludes the storage of more that one block at any computer without substantial rewriting. A second problem results from the static allocation of arrays used in Fortran. This forces the memory allocated to each block to be the size of the *largest* block in the entire domain. This waste of memory results from the lack of dynamic memory allocation; it is a severe restriction if attempting to optimize the use of memory with bin packing. With current system overheads, these restrictions limit the size of computation we are currently able to execute to approximately five million grid points on the 512 node Delta machine.

9 Example Simulation

The concurrent solution methods described in this paper have been validated using a complex shock interaction problem. The top figure in the first color plate illustrates experimentally observed steady state effects from a supersonic flow (Mach 3.17) over two inclined faces [3, 5][2]. The second color plate shows density and pressure plots at the crossection approximately half way through a simulated viscous flow over this geometry. These results were obtained on the Intel Delta machine. The results compare favorably with the experimental data: Notable aspects are the formation of the simple oblique and corner shocks, embedded shocks, and slip surface.

The lower figure on the first color plate shows the surface pressure at the crossection. Notice that the embedded shock is modeled without ringing due to the TVD scheme

[2]Republished with permission from the Journal of Computational Physics [3]

employed. There are 80 grid points along the Z-direction; thus, the embedded shock is smeared over approximately eight grid points. Our previous inviscid experiments with this problem signify a smearing of only 3 grid points around the analytical solution. We attribute this additional smearing to viscous effects.

10 Conclusion

This paper gives a status report on concurrent programming activities concerning an industrial strength Navier-Stokes code. This code is used by The Aerospace Corporation for complex multi-body launch vehicle configurations and nozzle flow calculations. The code uses a finite volume TVD scheme for computing both steady state and unsteady solutions to the 3-D compressible Navier-Stokes equations. It is second order accurate in space and is fully vectorized for operation on Cray computers. A line-by-line relaxation algorithm is used to accelerate the convergence for steady state solutions.

The code has been carefully validated using symmetric three-dimensional shock interaction problems for which there is both experimental data and analytical results. Some example calculations are shown here, however, the same problem has been used to validate numerous aspects of the code including turbulence modeling, irregular decompositions, multi-body configurations, and implicit techniques.

The concurrent formulation utilizes domain-decomposition techniques that allow the irregular computational grid to be decomposed into regular blocks of varying sizes. This is achieved by decomposing each axis independently. Multibody configurations are modeled through holes in the decomposition that serve simply to synchronize the concurrent algorithm. The basic scheme can be load-balanced through standard bin-packing algorithms, however, this has proved difficult due to limitations with Fortran style programming.

We are now applying the code to a complete implicit launch vehicle simulation containing 10 individual solid body components. We do not expect the current Intel Delta machine to be capable of conducting this experiment but expect other resources to be available by the time we have set up the simulation.

References

[1] Baldwin, B. S., and Lomax, H., *Thin Layer Approximation and Algebraic Model for Separated Flow.* AIAA Paper 78-257, 1978.

[2] Chandy, K. M., and Taylor, S., *An Introduction to Parallel Programming.*, Jones and Bartlett, 1991.

[3] Charwat, A. F. and Redekeopp, L. G., *Supersonic Interference Flow Along the Corner of Intersecting Wedges*, AIAA 5(3).

[4] Harten, A., *High Resolution Schemes for Hyperbolic Conservation Laws.* Journal of Computer Physics, volume 49, pp. 357, 1983.

[5] Shanka, V., Anderson, D., and Kutler, P., *Numerical Solutions for Supersonic Corner Flow*, JCP 17: 160-180, 1975.

[6] Wang, J. C. T.,, et al. *A Three Dimensional Finite Volume TVD Scheme for Geometrically Complex Steady State and Transient Flows.* AIAA Paper 88-0288, 1988.

[7] Wang, J. C. T., and Widhopf, G. F., *An Efficient Finite Volume TVD Scheme for Steady State Solutions of the 3-D Compressible Euler/Navier-Stokes Equations.* AIAA Paper 90-1523, June 1990.

[8] Wang, J. C. T., and Widhopf, G. F., *A High Resolution TVD Finite Volume Scheme for Euler Equations in Conservation Form.* Journal of Computer Physics, volume 84, pp.145, 1989.

[9] Wang, J. C. T., and Widhopf, G. F., *Numerical Simulation of Blast Flowfields Using a High Resolution TVD Finite Volume Scheme.* International Journal of Computers and Fluids, 18(1):103-137, 1990.

[10] Widhopf, G. F., and Wang, J. C. T., *A TVD Finite Volume Technique for Nonequilibrium Chemically Reacting Flows.* AIAA Paper 88-2711, 1988.

A Message-Driven Programming System for Fine-Grain Multicomputers[1]

Daniel Maskit and Stephen Taylor
Scalable Concurrent Programming Laboratory
California Institute of Technology

March 1, 1993

Abstract

This paper describes an experimental message-driven programming system for fine-grain multicomputers. The initial target architecture is the J-machine designed at MIT. This machine combines a unique collection of architectural features that include *fine-grain processes*, *on-chip associative memory*, and *hardware support for process synchronization*. The programming system utilizes these features via a simple message-driven process model that blurs the distinction between processes and messages: messages correspond to processes that are executed elsewhere in the network. This model allows code and data to be distributed across the nodes in the machine, and is supported at every stage of the program development cycle. The prototype system we have developed includes a basic set of programming tools to support the model; these include a compiler, linker, archiver, loader and microkernel. Although the concepts are language independent, our prototype system is based on GNU-C.

[1]The research described in this report is sponsored primarily by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Navel Research under contract number N00014-91-J-1986. The first author is partially supported by an NSF Graduate Research Fellowship.

1 Overview

The *Scalable Concurrent Programming* group is developing portable, high-performance programming systems that execute efficiently on scalable multicomputers [16]. The programming systems we have been involved in, PCN [6], Strand [12], and FCP [18], all share the same fine-grain process model. Briefly stated, the model has the following characteristics:

> A computation is a recursively defined collection of concurrent processes that may execute in any order or in parallel. Processes communicate and synchronize using the notion of *monotonicity*. Mapping is achieved using annotations, for example *foo(...)@n*, specifies that process *foo(..)* is executed at node *n*. Processes allocate and deallocate memory piecemeal when necessary and do not employ a stack; they may share global variables. A process may *suspend* at any time during its execution for the purpose of covering latency while relocating code or data.

Monotonicity is a simple, architecturally independent, shared variable method for specifying communication and synchronization. The concept allows the meaning of a program to be isolated from its environment. It can be used to provide termination detection, distributed data structures, monitor style atomicity, and a wide-variety of stream communication protocols [10, 12].

Multicomputer architectures have traditionally supported only Unix-style, coarse-grain, stack-based, processes. Thus our previous implementations have been forced to utilize an emulation technique to provide efficient systems [13, 18, 20]. Recently, two radical new architectural experiments have been conducted at Caltech and MIT. At Caltech, the Mosaic architecture, developed by the Submicron Systems Architecture Project, is designed expressly to support efficient fine-grain process execution [16]. The J-machine is a similar design, developed at MIT by the Concurrent VLSI Architecture Project, that supports fine-grain processes but also provides on-chip associative memory, and hardware support for process synchronization [8].

This paper describes an experimental message-driven programming system and its implementation on a 32-node J-machine. The techniques used in the construction of this system are language independent, however, the prototype system is based on GNU-C. The system employs a novel implementation strategy for fine-grain programs that has the following characteristics:

- New architectural features are directly accessed through native code compilation.
- Hardware performance is delivered directly to applications by removing software overheads associated with message-passing.
- Code and data distribution are provided by a simple run-time micro-kernel.
- Communication latency is hidden by a process suspension mechanism.

- Communication-oriented compiler optimizations are used to remove overheads associated with the copying of data structures.
- Processes utilize a heap-based allocation scheme rather than a stack and may thus suspend without copying overheads.

The implementation strategy employs a layered approach to compilation that is illustrated in Figure 1. The upper layer provides a source-to-source transformation system. This layer accepts programs written in terms of reusable abstractions; it generates high-level concurrent programs. These high-level programs are expressed as concurrent processes that communicate via shared monotone variables. The next layer translates high-level programs into a low-level systems programming notation. This notation provides only remote function invocation and forms the compiler target language. Programs and data expressed in the low-level notation may be distributed over the entire machine using a combination of linkage tools and a run-time micro-kernel.

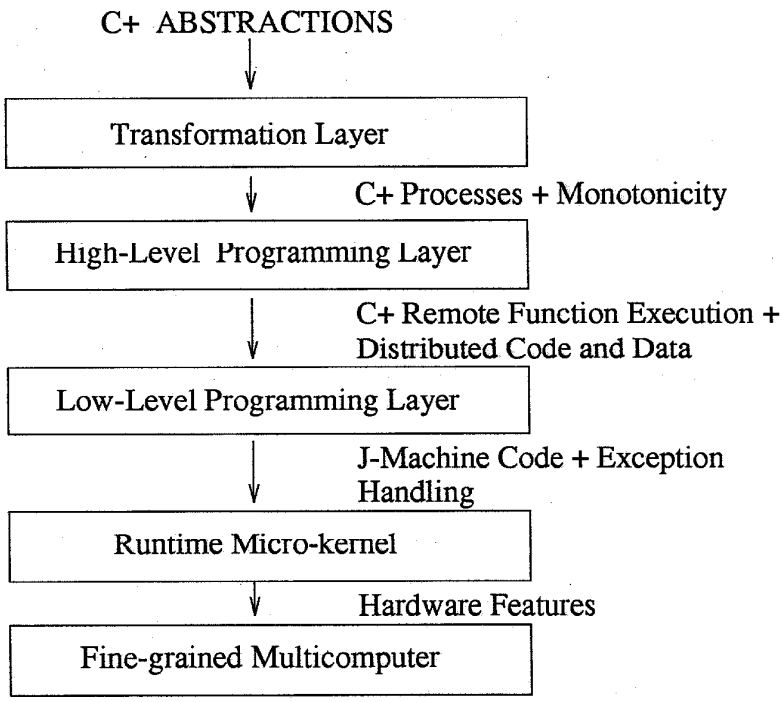


Figure 1: Layers in the Compilation Process

The prototype programming system implements all of the important aspects of this layering and includes a compiler, linker, archiver, loader and microkernel. It is currently being used for a variety of large-scale applications experiments in Computational Fluid Dynamics, Circuit Simulation, and Molecular Modeling. The source-to-source transformation layer is that designed for PCN programs [11]; thus, this paper focuses on the lower layers of the system.

2 Abstract View of the J-Machine

The programming system uses a variety of hardware features that are provided by the J-machine architecture. Figure 2 provides an abstract view of the hardware that highlights these features.

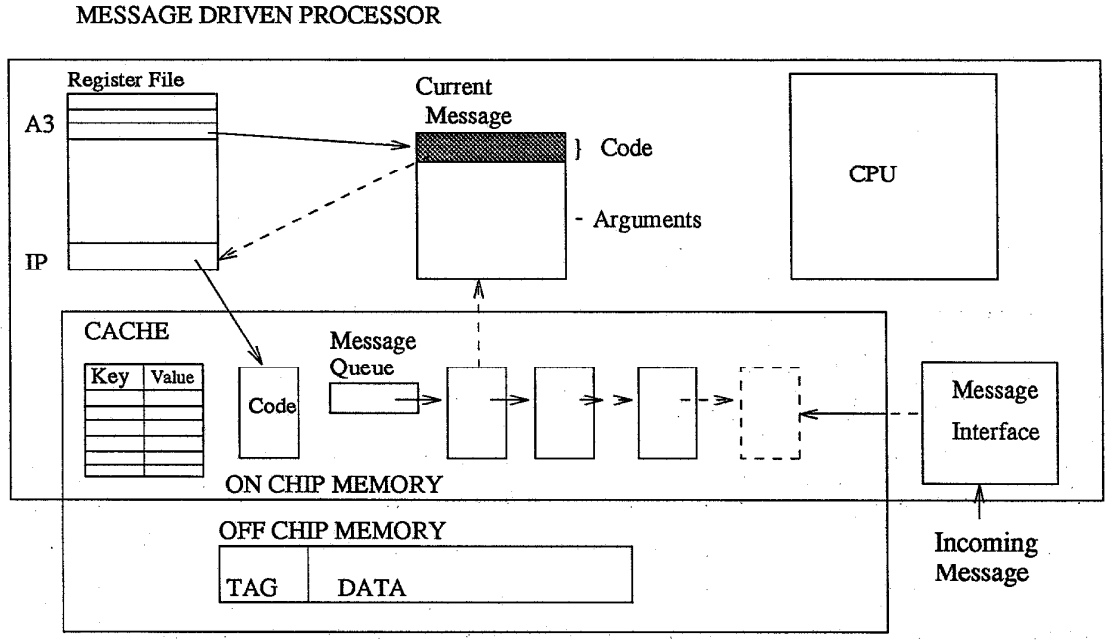


Figure 2: An Abstract View of the J-Machine

A message corresponds to a process in transit. Each message carries the address of the code to execute and the process arguments. Upon arrival, each message is appended to a scheduling queue that contains all active processes. A process is executed by removing it from the active queue, placing its code address into the instruction pointer, and placing a pointer to its arguments in machine register A3. The process then begins execution with its arguments residing in the original message, accessed via register A3.

The machine also provides an associative memory that allows a full 32-bit association between a key and its associated value. This memory can be manipulated via simple *enter*, *translate*, and *probe* operations. The *enter* operation adds an entry to the table at some index. The *translate* operation reads the data associated with an index. Finally, the *probe* operation inspects the table to determine if a location is in use. These functions allow the associative memory to be used as a hash table.

A final feature that is useful is a *tag* field associated with every memory word. This can be used to provide a simple process synchronization mechanism. The tag may be preset to an *undefined* state. If a memory word is accessed while its tag is set to *undefined*, for example during an arithmetic operation, an exception is generated and a fault handler executed.

3 Low-level Programming Layer

The low-level programming system model assumes a fully connected collection of n multicomputer nodes numbered *0* to *n−1*. This layer provides only sequential code execution and remote function invocation. A simple communication library is used to provide this functionality; it consists of just three functions:

nodes(). Returns the number of nodes in the allocated machine.

node(). Returns the identifier of the current node, an integer in the range 0 to *nodes()-1*.

spawn(n,f,l,p). Spawns the function named f on node n with two arguments. The pointer p is a reference to program data and l is the corresponding data length.

Figure 3 outlines the use of these functions for constructing low-level programs. Remote functions execute as independent concurrent processes and may invoke other functions at the current node during their execution. They may also cause new processes to be created at any node in the machine using the *spawn* library function. There is no synchronization involved in this activity: The calling function proceeds immediately without waiting for termination of the remote function f; moreover, the remote function does not return a value.

3.1 Example Low-Level Program

Program 1 shows a simple example of the use of the low-level programming system. The program begins execution at the keyword *main* on node zero. It then traverses the entire machine one node at a time in sequential order. At each node the program prints the node number multiplied by the number of times that node has been visited. Thus on a three node multicomputer, the output would be:

0 1 2 0 2 4

3.2 Implementation Techniques

The implementation of this layer in the system builds upon a GNU-based C compiler that has been retargeted for the J-machine. The message-driven process model of the machine is utilized directly for remote function invocation. The spawn library call initiates the sending of a message and transmits data from memory directly into the network;

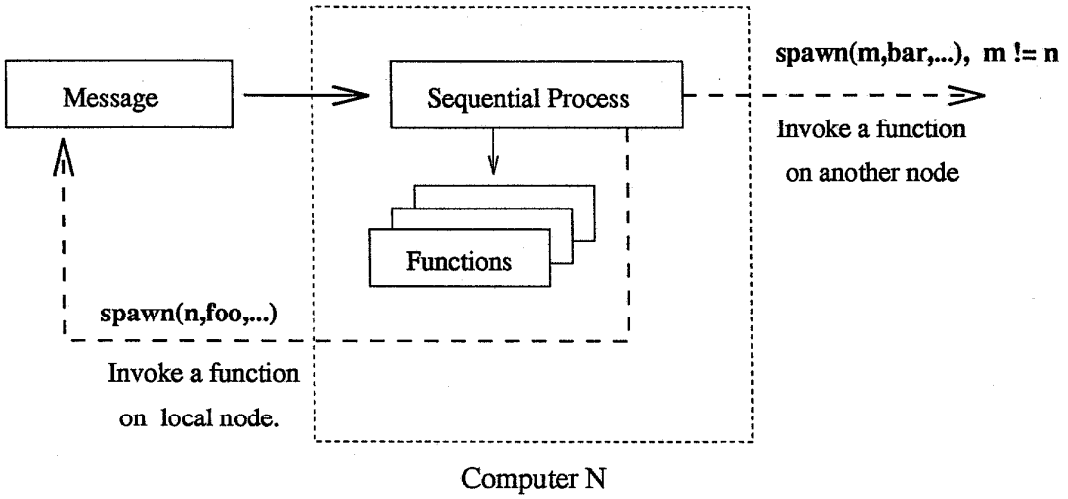


Figure 3: Single Node Operation

```
#define NUM_REPS 2

main()
{   int rep = 0;
    Hop(sizeof(rep), &rep);                  /* Local call */
}

void Hop(int size, int* rep)
{   int next = (node()+1) % nodes();         /* Where next */
    if(node() == 0) (*rep)++;
    if(*rep <= NUM_REPS) {                   /* Not done ? */
        printf("%d ",node()*(*rep));         /* Print node number */
        spawn(next,Hop,sizeof(int),rep);     /* Hop on */
    }
}
```

Program 1: Example Low-Level Program

thus there are no copying overheads associated with message sending. The compiler has been modified to allow function arguments to reside either in node memory or a received message. Thus, there are no copying overheads associated with receiving a message.

To enhance system performance, functions are separated into two groups: those that are distributed across the entire machine and those that are replicated at every node. Replicated functions are frequently used code segments such as floating point routines, the micro-kernel itself, and critical application functions. Distributed functions are intended to be used infrequently or at only a small subset of the nodes. The compiler generates code that forces micro-kernel intervention during function invocation if the appropriate code is not resident at the current node.

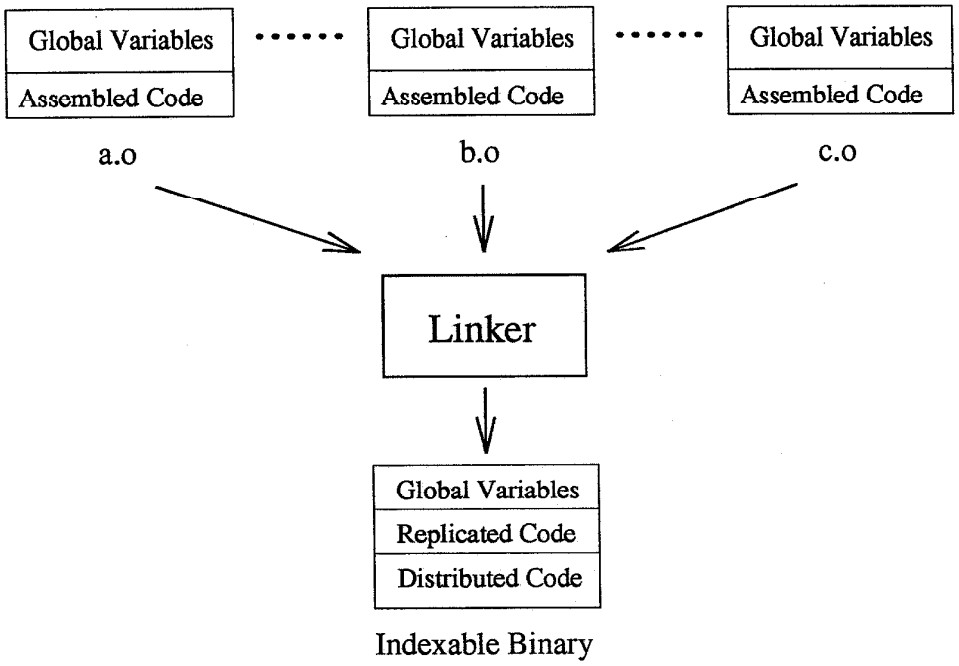


Figure 4: Linkage and Loading Stages

Figure 4 shows the structure of compiled code submitted to the linkage stage of the compilation pipeline. The linker accepts a collection of object files and associated libraries; it creates an indexable binary file that can be partitioned by the loader into images to be loaded into each node. The linker assigns each distributed function to a *home node* in the machine. A simple bin-packing algorithm is used to balance the quantity of code resident at each node; There is a standard interface to the linker that allows experimentation with alternative code mapping algorithms. In our current implementation, global variables are replicated at every node and coherence is not maintained between nodes.

After code mapping is complete, the linker assigns unique identifiers to all functions. If a function is invoked during program execution, the micro-kernel uses these identifiers to locate the appropriate code. The identifier is a pair of the form:

< *logical-node, function address* >.

The *logical-node* entry signifies the home node of the code, and the function address specifies the location of the code at the home node. The translation from logical-node to physical-node is made by the loader after allocation of a partition of the machine.

```
SPAWN_HANDLER:
    if (function replicated OR this is home node OR in hash table)
        invoke function
    else {
        if(not requested already)
            send REQUEST message
        suspend process in queue associated with function
    }

REQUEST_HANDLER:
    Send function to requesting process in RECEIVE message

RECEIVE_HANDLER:
    Allocate storage for function code
    Copy code from message to storage
    Enter function in hash table
    Wake processes waiting for function
```

Program 2: Micro-kernel Message Handlers

Using function identifiers, the microkernel can be defined as a collection of message handlers as shown in Program 2. The *spawn* handler is responsible for locating the code of a function and subsequently executing it. It is used to handle both spawn messages, originating at other nodes, and local function invocation. A function can be executed immediately if it is either replicated or it is distributed and the current node is the home node. Otherwise, the code is either in a local hash table or must be fetched from its home node and deposited into the hash table. The hash table is implemented using the available associative memory provided by the J-machine hardware. If code is requested from another node, then the current process suspends until this code is copied locally. The *request* handler deals with a request for code from another node and transmits the code from a known location determined using the code's unique identifier. Finally, the *receive* handler is invoked when requested code is received. It wakes all processes that are suspended awaiting code reception and updates the local hash table to store the received code. The hash table is purged to free space when necessary.

4 High-level Programming Layer

The purpose of the high-level programming layer is to hide the underlying low-level system and add two concepts to sequential programming: *concurrent process execution* and *monotone variables*. Concurrent processes are used in the definition of concurrent algorithms, and to cover latency by multiprocessing at a single node. Recall that monotone variables are an abstract, architecturally independent method for expressing communication and synchronization. In coupling these concepts we have laid a foundation for the formal treatment of the resulting concurrent programs [6].

These concepts have been used extensively in practical concurrent programming experiments using the PCN, Strand, and FCP systems. A study of the resulting application codes yielded a collection of generic programming techniques that are repeatedly reused and combined in different guises. These techniques form a core set of tools with which to build complex applications. They allow a wide variety of stream communication protocols, distributed data structures, monitor style atomic operations, and termination detection schemes to be implemented without complicating the language semantic. A complete exposition of the techniques can be found in Chapter 3 of [12].

All stream communication protocols are denoted using list operations, without adding new concepts to the language semantic. Thus, sending a message is denoted as adding an element to the beginning of a list e.g. *S=[message(...)|Ss]*. Receiving and destructuring a message is denoted using a simple matching operation denoted *S?=[message(...)|Ss]* [6].

4.1 Example High Level Program

Program 3 shows a generic producer-consumer program that illustrates many of the programming concepts used in the construction of concurrent algorithms. Execution begins in node 0 at the main function. Initially, a stream is created to carry messages from the producer to the consumer (1) and the global array A is initialized (2). Subsequently, two concurrent processes are spawned: The first is a producer that executes at node 0 (3); The other is a consumer that executes at node 1 by virtue of the mapping annotation @ (4). Notice that the producer and consumer share the communication stream S. Only when both processes have terminated, does the tidyup function execute (5). Upon its termination the entire program terminates.

The producer sends a single message to the consumer containing a copy of the array A (6). It then recursively sends n−1 further copies of the array (7) until termination occurs and the stream is closed (8). The consumer *suspends* until a message is received by virtue of the matching operation (9). It subsequently uses the message (10), and finally consumes any remaining messages recursively (11). Eventually, the consumer terminates when the stream is closed.

4.2 Implementation Techniques

There are many ways to implement the concept of monotonicity. In previous systems, a simple implementation technique has been used in which the first occurrence of a vari-

```
int A[50];

main()
{   stream S;                          /* 1 */
    initialize(A);                     /* 2 */
    {| | producer(10,S);               /* 3 */
        consumer(S)@1;                 /* 4 */
    }
    tidyup();                          /* 5 */
}

producer(int n, out stream S)
{   if(n > 0) {
        S = [A | Ss];                  /* 6 */
        producer(n−1,Ss);              /* 7 */
    }
    else
        S = [];                        /* 8 */
}

consumer(in stream S)
{   if(S ?= [A|Ss]) {                  /* 9 */
        use(A);                        /* 10 */
        consumer(Ss);                  /* 11 */
    }
}
```

Program 3: Generic Producer-Consumer Protocol

able is placed at a single location within a multicomputer. All subsequent occurrences are represented by inter-processor references. Variables are initially *undefined* and process synchronization is achieved by suspending processes until the value of a variable is known. This suspension mechanism associates the process with the variable or remote-reference until the value can be ascertained. Stream protocols are implemented by structure copying between processors. The language semantic is specifically constructed to allow this copying to be achieved asynchronously and without sophisticated communication or locking protocols [18].

For fine-grain multicomputers exactly the same implementation techniques can be used as on medium-grain machines. As a result, the PCN system has been retargeted to the J-machine. However, by knowing the form of the generic programming techniques, it is possible to construct new *communication-oriented compiler optimizations* that capitalize on the structure of the technique in use. These optimizations take advantage of the hardware without changing the basic programming semantic. For example, in Program 3 additional information in the form of an abstract data type *stream* has been provided. This signifies to the compiler that a particular stream protocol is in use. Further, from the function prototypes, it may be determined that the direction of communication is from the producer to the consumer. This information makes it possible to generate code that allows the processes to communicate without an explicit representation of the list structure avoiding all overheads associated with structure copying. The list notation only signifies the constraint that message order is preserved.

Figure 5 outlines how this protocol is implemented using the J-machine tagging structure and a collection of message handlers. The handlers are trivial to implement using the systems programming layer described in Section 3. In this figure, the producer is signified by the process P and the consumer by the process C.

The concurrent processes are compiled into two calls to the spawn function. The first is spawned at node 0 (the current node) and simply invokes the producer; The second causes the consumer to be spawned at node 1. In order to establish the shared stream S, the producer is suspended until the location of S at the consumer in node 1 becomes *known*. Similarly, the consumer suspends until the first message produced by the producer is *known* (Λ).

Eventually, the stream location becomes *known* by virtue of communication sent from node 1 and the producer is then scheduled (B). The producer now begins to send messages to the consumer. The first message causes the consumer to be scheduled (C). If further messages arrive before that consumer is executed they are simply associated with the stream variable (D). Eventually, the consumer becomes the current process by reaching the front of the message queue. At this time it may, using a tail recursion optimization, iteratively consume all of the available messages and suspend to await further messages.

A simple counting mechanism is used to provide a barrier that detects termination of the two concurrent processes. A counter is associated with entry to a parallel block, this counter is initialized to the number of processes spawned. Both the producer and consumer are provided with the location of this counter. When a process terminates it uses communication to decrement the counter. Eventually, termination is signified by a counter value of zero.

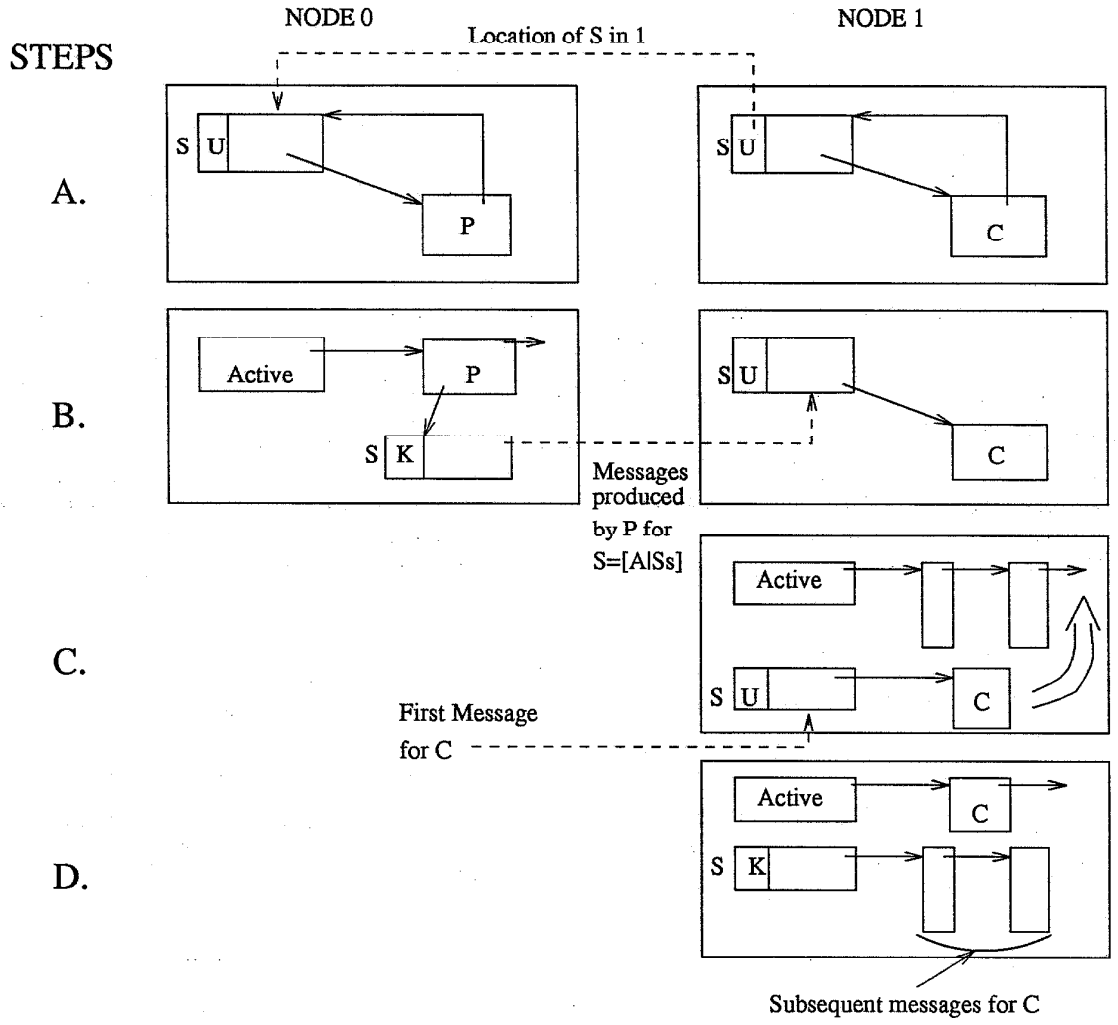


Figure 5: Stream Communication

All of the operations we have described involve only one way communication. Further, all of the synchronization operations can be built on top of variables that are in some binary state *known* or *unknown*. This form of synchronization is precisely that provided by the tagging structure of the J-machine.

The producer-consumer protocol is the simplest protocol to implement. However, minor generalizations of the basic implementation strategy described here allow all of the other basic stream based programming techniques to be implemented. These implementation strategies are the subject of ongoing research.

5 Related Research

The basic concept of message-driven process execution was conceived by Chuck Seitz and incorporated in a variety of reactive programming systems [16, 17, 2]. This concept has been a recurrent theme in both hardware and software designs from the Submicron Systems Architecture Project over the years. It has been incorporated directly into most commercial multicomputer programming systems. In common with Cantor and Actor [1], programs we utilize message-driven concurrent processes that do not employ a stack. However, unlike these systems, our processes communicate and synchronize via a simple, portable, monotone variable concept, suspend to cover latency, and share global state for efficiency.

Recently, a new generation of fine-grain systems have appeared. The MADRE [4] system developed for the Mosaic Architecture [16] attempts to distribute the micro-kernel across multiple nodes. It provides the ability to utilize resources, such as message buffers, at other nodes when local resources are exhausted. We view programs that exhaust local buffers as poorly conceived and do not seek to encourage this programming practice. Thus we deal with message overflow by allowing the network to block until resources are available. This alternative provides graceful degradation of performance without substantial complexity.

A Concurrent Smalltalk (CST) [15] system has also been developed for programming the J-machine. CST requires a larger and more complex micro-kernel to handle a broad range of language concepts that our applications do not require.

The basic compiler and run-time techniques associated with the distributed implementation of our fine-grain process model were developed in early work on FCP [18]. Perceived improvements were incorporated into *Strand* [12, 10] and copied directly into PCN [6] without change.

The *Strand* programming system integrated sequential and concurrent programming concepts via a complex, ad-hoc, interface. This allowed standard C and Fortran compilers to be used for sequential subroutines [10]. The PCN system simplified this interface by sharing data types between concurrent and sequential components of a program. The research described in this paper represents the next step in this direction: we no longer make a distinction and apply native code compilation techniques to both components. We have also abandoned the concept of *non-deterministic choice* [10] as it was found to be irrelevant in practice. We utilize communication-oriented compiler optimizations to pro-

vide efficient implementations of frequently used programming techniques; this continues work described in Chapter 7 of [18].

The abstract notion of monotonicity is not new. We use the term to distinguish a portable, shared variable, method for defining communication and synchronization [12, 10]. The concept has theoretical importance in that it allows the meaning of a program to be isolated from its environment. Similar concepts have appeared in functional programming through the notions of referential transparency [3, 5] and single assignment variables [9, 19]; logic programming through the logical variable [20]; reactive [16] and actor [1] programs through message-passing. However, these broad generalizations are not particularly insightful: Innocuous and subtle changes in semantic drastically impact the usefulness of the concept. These may result in inefficient concurrent implementations of our fine-grain programming model or restrictive programming techniques. To give a flavor for some of the practical considerations:

- The simple assignment operation $X = Y$ where, both X and Y are variables, allows *aliasing*. Although problematic for dependency analysis, if defined correctly, this concept is easy to explain, trivial to implement, and enables many useful parallel programming techniques. These include distributed data structures, termination detection schemes, predicate implementations, and broadcasting protocols.

- The assignment $X = f(X)$ generating circular data structures results in complex distributed implementations, however, it is rarely used in practice. In contrast, recursive data structures, as in $X = f(f(Y))$, provide a simple method for describing and manipulating complex messages.

- General unification operations such as $f(X, X) = f(a, a)$ require complex locking protocols on parallel machines. Problems such as deadlock, livelock and starvation must all be considered. However, this complexity can be replaced with simple asynchronous parallel algorithms if pattern matching is used to manipulate messages. The generality of full unification is largely of use on contrived problems.

- The semantics of boolean testing operations and the placement of assignment operations can make substantial impact on the performance of suspension mechanisms and other process structures.

The various tradeoff's have become apparent only after many years of study both in the construction of concurrent applications and the implementation of programming systems that employ the fine-grain process model.

6 Conclusion

The main concepts described in this paper have been implemented and are in use on a prototype J-machine at Caltech. The model we have advocated for concurrent programming is not new and has been used extensively for applications development in a variety of other systems. The contribution of this work rests on new implementation techniques for

fine-grain architectures. The layered system we have proposed provides a clear separation of concerns and adds only a few simple changes to existing sequential languages.

The implementation techniques allow new architectural features to be accessed directly via native code compilation. Hardware performance is delivered directly to applications by removing software overheads associated with message-passing. Messages are copied directly into the network on sending, and processes execute directly out of message buffers on receiving. Code and data may be distributed through a combination of linkage and run-time micro-kernel support. Communication latency is hidden by a process suspension mechanism and communication-oriented compiler optimizations are used to remove overheads associated with the copying of data structures. Our processes utilize a heap-based allocation scheme rather than a stack and thus may suspend without copying.

Although only a single stream communication protocol has been described in this paper, a number of other protocols can be implemented using simple extensions to these ideas. These optimization techniques are the subject of our ongoing research. In addition, we seek to support complex programming abstractions directly in a C-like syntax rather than through compilation of an intermediate language such as PCN.

7 Acknowledgments

This work owes a great debt to other members of the research groups at both the California Institute of Technology and the Massachusetts Institute of Technology. In particular, Yair Zadik and Chris Ziomkowski implemented the linker, loader, archiver, and floating-point support. Dong Lin collaborated on the initial version of the compiler. Andrew Chang, Mike Noakes, and other members of the Concurrent VLSI Architecture project at MIT have provided constant assistance with low-level software and hardware.

References

[1] Agha, G., *Actors*, MIT Press, 1986.

[2] Athas, W. C. and Seitz, C. L., Cantor User Report, Version 2.0, Tech. Rept. 5232:TR:86, Department of Computer Science, California Institute of Technology, 1986.

[3] Backus, J., Can Programming be Liberated from the von Neumann Style? CACM, 12(8), August 1978, pp 613-41.

[4] Boden, Nanette J., Runtime Systems for Fine-Grain Multicomputers. Ph.D. thesis, California Institute of Technology, 1993.

[5] Burnstall, R. M., MacQueen, D. B., Sannella, D. T., HOPE: An Experimental Applicative Language, Lisp Conference Records, Stanford University, Stanford, California, 1980, pp 136-43.

[6] Chandy, K. M., and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

[7] Chern, I., and Foster, I., Design and parallel implementation of two methods for solving PDEs on the sphere, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991.

[8] Dally, W. J., et al., The J-Machine: A fine-grain concurrent computer, Information Processing 89, G. X. Ritter (ed.), Elsevier Science Publishers B.V., North Holland, IFIP, 1989.

[9] Davis, A. L. and Keller, R. M., Data Flow Program Graphs, IEEE Computer, vol 13, pp48-56, Nov, 1980.

[10] Foster, I., Kesselman, C., and Taylor, S., Concurrency: Simple concepts and powerful tools, *Computer Journal*, 33(6), 501-507, 1990.

[11] Foster, I., and Stevens, R., Parallel programming with algorithmic motifs, *Proc. Intl Conf. on Parallel Processing*, Penn. State Univ. Press, 1989.

[12] Foster, I. and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1989.

[13] Foster, I., and Taylor, S., "A Portable Run-Time System for PCN", Argonne National Laboratory, Technical Memorandum No. 137, ANL/MCS-TM-137, January, 1990.

[14] Harrar, H., Keller, H., Lin, D., and Taylor, S., Parallel computation of Taylor-vortex flows, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991.

[15] Concurrent Smalltalk on the Message-Driven Processor, Masters Thesis, MIT, Sept., 1991.

[16] Seitz, C. L., Multicomputers, *Developments in Concurrency and Communication*, C.A.R. Hoare (ed.), Addison-Wesley, 1991.

[17] Su, W., Reactive-Process Programming and Distributed Discrete Event-Simulation, Ph.D. Thesis, California Institute of Technology, Caltech-CS-TR-89-11, 1990.

[18] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[19] Tesler, L. G., and Enea, H., A Language for Concurrent Processes, Proc. SJCC, AFIPS Press, 1968.

[20] Warren, D.H.D., Applied logic — its use and implementation as a programming tool, SRI International Tech. Rep. 290, 1983.

Immersing the Scientist in Data[1]

Understanding Irregular Three-dimensional Scalar Fields Through Immersion in a Virtual Environment

Michael E. Palmer, Alan H. Barr, and Stephen Taylor
Scalable Concurrent Programming Laboratory
California Institute of Technology

March 9, 1993

Abstract

This paper describes a concurrent algorithm for scientific visualization of irregular, three-dimensional scalar data in an interactive virtual environment. The algorithm is designed to be efficient at direct volume rendering of consecutive frames from nearby viewpoints, such as those arising from smooth head motion or required to display left and right stereo frames.

The main contributions of this work are scalable methods to dynamically partition irregular scalar data for ray casting, to efficiently redistribute this data as the dynamic partitioning changes, and to load balance the rendering computation.

The concurrent algorithm is intended to be scalable to multicomputers containing many thousands of nodes. It is structured around fine-grain parallelism and low-latency communication. Each node is responsible for a contiguous area of the screen and the volume of screen-space directly behind it. The volume allocated to a node changes only slightly with slight changes in viewpoint, and therefore little redistribution of data is usually needed from frame to frame. Furthermore, ray casting can be effected without communication. The computation can be load balanced when necessary by shifts in the assignment of screen area to nodes.

[1]The research described in this report is sponsored in part by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Naval Research under contract number N00014-91-J-1986. The reseach is also supported by an NSF-PYI Award under contract ASC-9157650 with industrial matching funds from Intel Supercomputer Systems Division and Sun Microsystems Corporation.

1 Introduction

The complex and irregular three-dimensional computations now possible on modern multicomputers are swamping scientists with data; there is too much information to absorb and synthesize into understanding. Currently, scientists scan huge piles of printed output, display two-dimensional cross sections of three-dimensional data, or reduce the volume of information through statistics. These methods may hide interesting details of the data; they fail to exploit the sophisticated apparatus for processing visual information that has evolved in humans. This apparatus allows humans to understand complex three-dimensional, shadowed, colored, and moving environments.

A powerful and intuitive alternative is to immerse the scientist in a running concurrent computation using an interactive virtual environment. Recent advances in multicomputer technology provide the power to simulate virtual environment representations of large computations. The goal of this research is to develop software concepts and strategies now that will be both useful on currently available hardware, and scalable to machines that will be available in the next few years. The visualization tools developed will allow the exploration of a variety of irregular concurrent computations that are under investigation in our laboratory. These include moving boundary problems in Computational Fluid Dynamics (CFD) and real time Magnetic Resonance Imaging (MRI) combining images at multiple resolutions. Our primary concern is to aid the understanding of these computations rather than to render realistic images.

Figure 1 illustrates the proposed hardware configuration in its final form. A concurrent rendering algorithm manipulates three-dimensional scalar data from an MRI scanner, another concurrently executing computation, or snapshots of a computation stored on disk. The scientist wears a helmet that presents a color display to each eye and tracks head movements. The rendering algorithm continually calculates, and displays to each eye, a two-dimensional view of the current volume of data appropriate to the current head position.

The most important factor in maintaining the illusion of reality in a virtual environment is sufficiently high frame rate. The brain can tolerate only a small time interval between a movement of the head and an appropriate update of the view. If this delay increases past a certain threshold, the user's trust in the supposed reality of the objects before him or her is shattered.

The ability to render three-dimensional data interactively at realistic frame rates is within the power of the next generation of massively parallel multicomputers. This advance has been made possible by improvements in communication hardware, such as wormhole routing [Seitz 91], [Dally 86], and the ability to distribute a framebuffer across the end-plane of a multicomputer. Rendering algorithms that are scalable to such multicomputers may be the first to provide interactive rates for direct volume rendering of very large, irregular datasets.

Virtual environment simulations request only a constrained sequence of views due to the continuity of head position and the similarity between views required for the left and right eyes. The concurrent methods presented in this paper exploit these constraints to

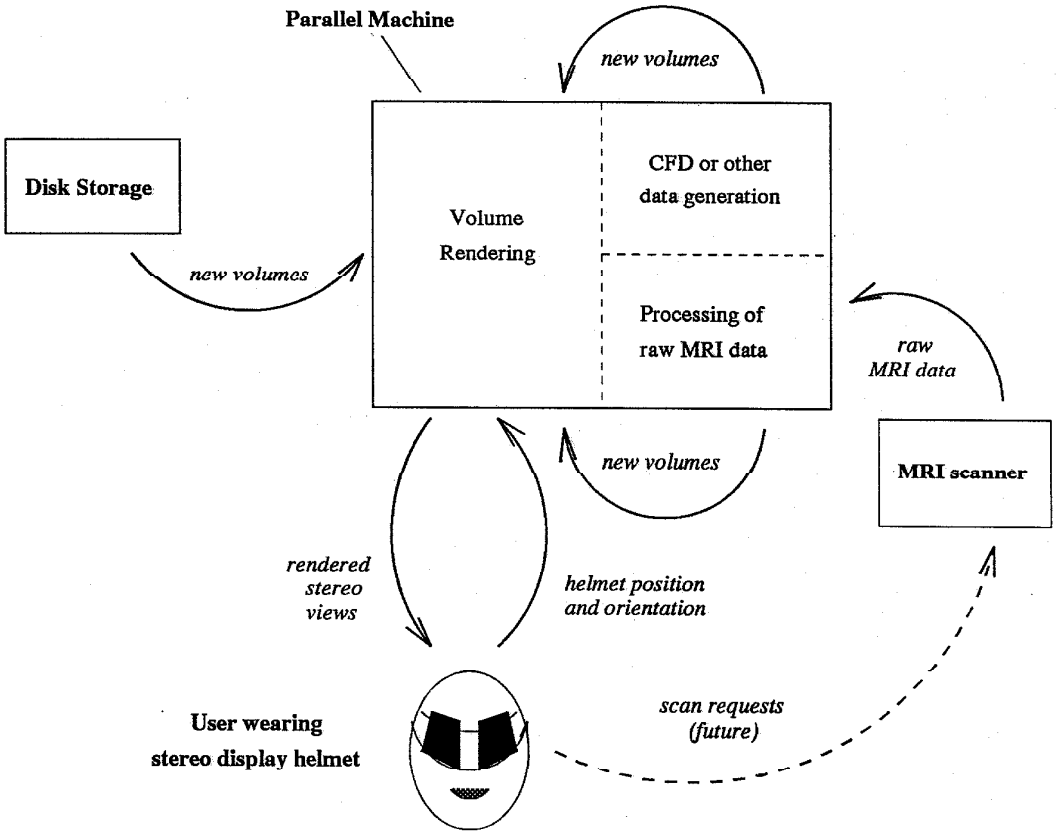


Figure 1: System Overview

minimize communication and utilize a straightforward load balancing scheme. The algorithm is structured around fine-grain parallelism and low-latency communication and is scalable to multicomputers containing many thousands of nodes. Each node is responsible for rendering a contiguous area of the screen and the volume of screen-space directly behind it. Once the data has been properly distributed, ray casting can be effected entirely within a single node. Further, small changes in viewpoint entail only small shifts in the ownerships of screen-space volume, and therefore require little communication. The computation can be load balanced when necessary by slight shifts in the assignments of screen area to nodes.

2 Overview of the Concurrent Volume Rendering Algorithm

The term "volume rendering" refers to a wide range of techniques for displaying representations of three-dimensional data on a two-dimensional screen. These techniques can be divided into those intended to display vector fields and those intended to display scalar fields. The algorithm presented here is designed to render scalar fields defined on irregular grids. Irregular grids arise, for example, in Computational Fluid Dynamics problems involving moving boundaries and Magnetic Resonance Imaging that combines images at multiple resolutions.

Scalar field visualization techniques can be further classified into those that extract a set of polygons from the data [Lorensen 87], [Gallagher 89], and those that render the scalar volume directly [Sabella 88], [Upson 88]. The latter technique treats the scalar volume as a semi-transparent cloud of varying density, illuminated by a lighting model, through which rays are cast to calculate an accumulated opacity and color [Blinn 82], [Kajiya 84], [Max 86].

The algorithm described here performs ray casting on irregular scalar volumes and is implemented by a collection of communicating processes. Figure 2 illustrates its main components. It accepts a sequence of data sets representing the three-dimensional volume at increasing timesteps and, at a higher frequency, a sequence of viewpoints. The algorithm continually computes the two-dimensional representation of the most recently loaded timestep of data from the current viewpoint.

A continuous scalar field can be described approximately by a finite number of irregular cells which fill the volume under consideration without overlap. This work uses convex polyhedral cells for simplicity, but the results of this paper will apply to other representations of semi-regular and irregular scalar fields.

The processes receive sets of irregular cells describing three-dimensional scalar volumes from either a running computation, or from checkpoints stored on disk. They also receive view requests from the user. Each process must first *transform* the original data-space coordinates of the irregular cells into screen-space coordinates. In this coordinate system, rays from the eye of the viewer point directly in the negative Z direction; the Y direction is taken to be the viewer's natural up direction.

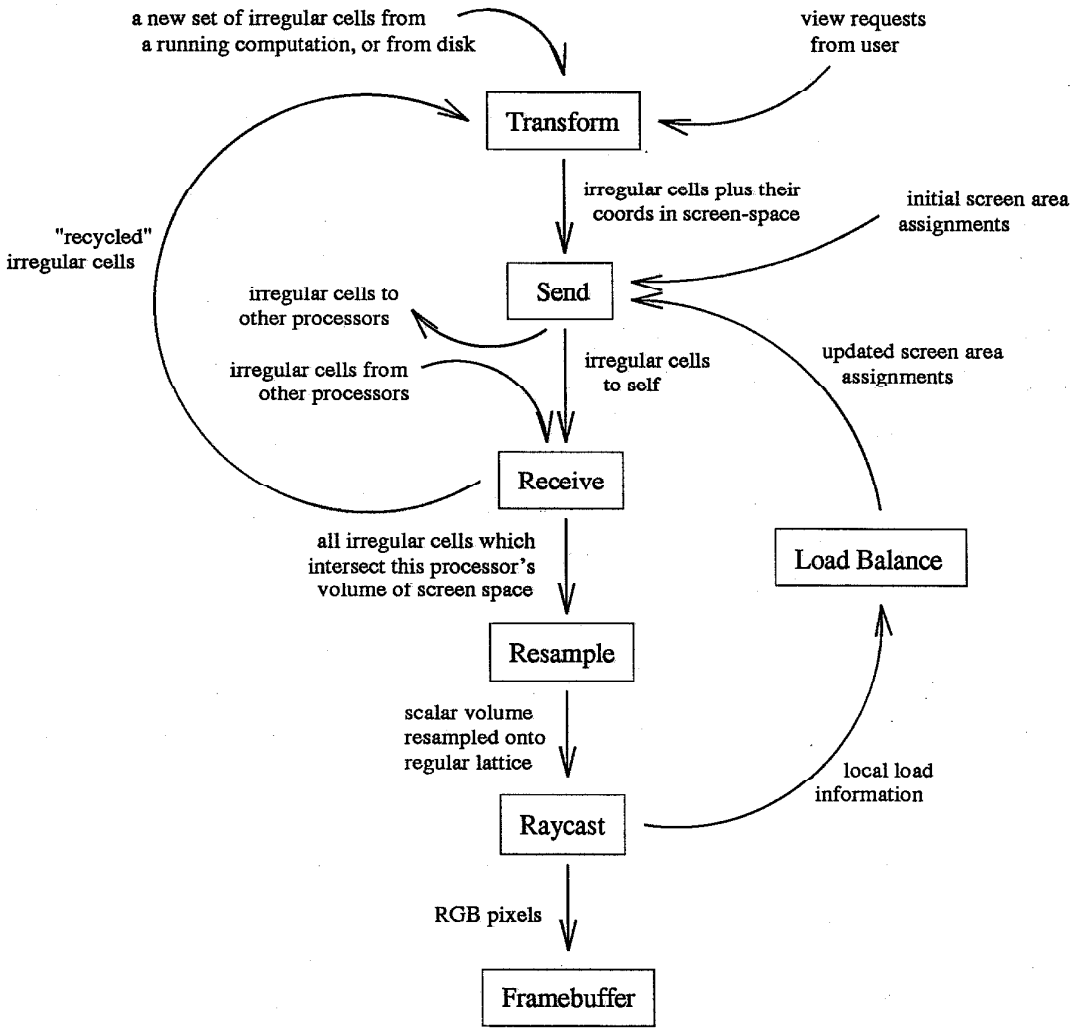


Figure 2: Overview of the Parallel Rendering Algorithm

The display screen is divided into initially equal rectangular areas and each process is assigned responsibility for rendering one of these areas. To render an area of the screen, a process must have a copy of all irregular cells that intersect the volume of screen-space lying directly behind that area. As the cells are initially loaded, processes will not, in general, have all the required cells. Thus each process must *send* the transformed cells to all appropriate processes. The processes then *receive* all cells required to render their respective screen areas. They may then complete rendering without further communication.

To complete rendering, a process first *resamples* the irregular cells onto a regular three-dimentional lattice, in order to facilitate ray casting. Each process next applies a color map and an opacity map to the scalar values in the regular lattice and *ray casts* by sending rays through the lattice directly in the negative Z direction. These rays accumulate a set of pixel color values which are sent to the *framebuffer* hardware for display.

After a viewpoint has been rendered, most data is held by the correct process for rendering the next viewpoint if it is not far removed from the previous one. The partitioning of screen-space will not change significantly for rendering a nearby viewpoint. Thus, unless a new volume of data has been loaded, most cells may be recycled, without communication, for rendering the next viewpoint. If a new set of irregular cells (e.g., the next timestep of a running computation) has been loaded, then the old cells are simply discarded.

After ray casting, each process can determine how much work it has performed to render its area of the screen. As each new view is requested, this local load information is passed to an algorithm that computes an improved *load balance*. The load balancing algorithm updates the assignments of screen area to processes so that they are no longer equal rectangles but, rather, enclose equal computational load. This requires only that the updated positions of the dividing lines between areas be communicated; on the next rendering cycle the appropriate cells will automatically migrate, and the load balance will be improved.

3 Mapping Volumes of Screen-Space to Processes

Recall that each process is responsible for calculating the pixel values for a rectangular area of the display screen. In order to do this, the process must know the values of all cells intersecting the volume of screen space directly behind its rectangle. The process can then cast rays straight back through the scalar volume data without the need to communicate with other processes. The assignment of volumes of screen-space to processes is dynamic: this allows the work load to be balanced as the viewpoint changes and the distribution of computational expense moves in relation to screen-space.

Figure 3 illustrates the ownership of areas of the screen and of volumes of screen-space. Here there are $n = 8$ processes, numbered from zero to seven. The left of the figure shows assignments of screen areas to processes, while the right shows the corresponding assignments of volumes of screen-space to processes. A process owns the entire volume

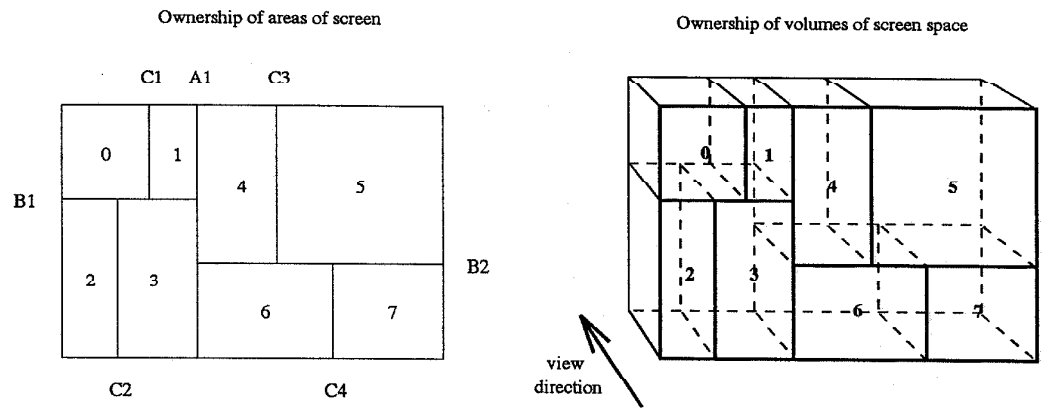


Figure 3: Dynamic Ownership of Volumes of Screen-Space

of screen-space behind its piece of screen area. In this example, the entire screen is recursively divided by log n sets of lines: first by the line A1, then by the lines B1 and B2, and finally by the lines C1, C2, C3 and C4. This representation of the tiling of the screen is memory efficient; each process need store only the positions of log n lines.

In order to ray cast an area of the screen, a process must have a copy of all cells that intersect its volume of screen-space. The initial copy of each cell is called the *master* copy. Some cells fall across the boundary between adjacent volumes, as shown in Figure 4. In such cases, *shadow* copies of the cell must be generated and sent to the appropriate processes. It is straightforward to integrate this procedure into the cell distribution algorithm.

4 Cell Distribution Algorithm

When the user's viewpoint changes, ownership of volumes of screen-space will such that the process that owned a given cell may have to pass the master copy of that cell to another process. We prefer to avoid having each process maintain a list specifying the volumes of screen-space owned by each process. Instead, the screen area is recursively cut by log n sets of lines. In Figure 3 there are three sets of lines: the A set, consisting of line A1; the B set, consisting of lines B1 and B2; and the C set, consisting of lines C1, C2, C3, and C4. To reach its final destination, a cell must hop through log n processes; at each step, a decision is made by that process that determines to which side of the appropriate line the cell propagates. The left side of Figure 5 illustrates the binary tree through which the cells travel, for $n = 8$. After log n decisions and hops to the indicated processes, the cell arrives at its correct owner. If it is found that a cell overlaps both sides of a given line, then it is straightforward to send one copy to one side of the line, and a shadow copy to the other side. At the leaves of the tree, all processes have a copy of all cells that

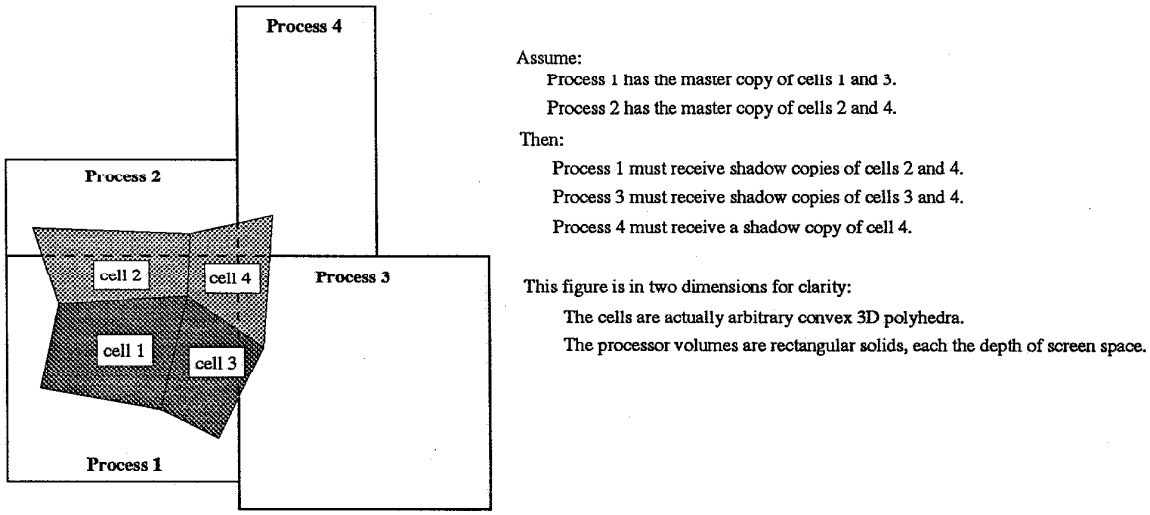


Figure 4: Master and Shadow Copies of Cells

intersect their volume of screen-space.

Some cost is incurred in the transmission of cells to new owners, as a cell must traverse $\log n$ processes to reach an arbitrary destination; however, this representation the tiling is memory efficient: each process need only store the positions of $\log n$ lines, rather than representations of n rectangles. The lines that must be stored by each process, for $n = 8$, are shown on the right of Figure 5.

5 Load Balancing

Load balancing is achieved by the movement of the lines dividing areas of the display screen, such as lines A1, B1, B2, C1, C2, C3, and C4 in Figure 3. In order to have consensus, a single process is responsible for deciding the position of each line. Simple algebraic formulas determine the "parent" of each line and its two "children". To the left of Figure 5 for example, process 0 is the parent of line A1; its two children at level A are itself (process 0) and process 4. These two processes are the parents, at level B, of lines B1 and B2, respectively; and so on.

Reports of workload travel up the tree by special messages. At the leaves of the tree to the left of Figure 5, a process' local load is determined by some measure of the computational expense it took to render the most recent frame. All eight processes pass this load up to their parent at level C. The parents at level C (processes 0, 2, 4, and 6) sum the loads of their children to compute their loads at level B. These are passed to their parents at level B (processes 0 and 4). Processes 0 and 4 sum these to compute their loads at level A, which are passed up to process 0, the parent at level A.

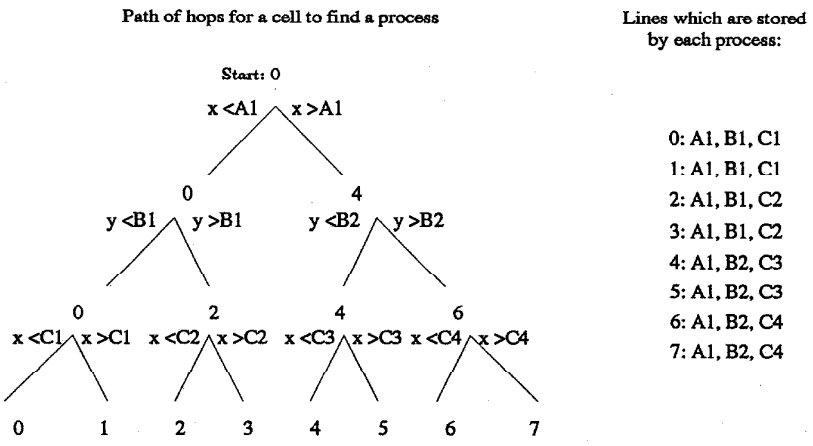


Figure 5: Redistibution of Cells through Hopping

A parent at any level will move the position of its line appropriately if the workloads of its two children are significantly different. Updated line positions travel back down the tree along with the reception of a new view request. When the user requests a view, the host process sends the request to process 0, which corresponds to the root of the tree. This initiates the cascade of messages down the load balancing tree. These messages include the new view matrix, and line position information, which is updated as it passes through the parents at different levels down the tree. When they are received by the leaf processes at the bottom, the new line position information is complete and personalized for each process; for $n = 8$, each process will receive the position of the three lines shown to the right of Figure 5.

There is no single trigger for a global load balancing operation; each time a new view is distributed, each parent at each level decides locally whether to change the position of its line, based on the load of its children and whether lines above it have been moved.

Although, conceptually, the algorithm operates as described, it is possible to apply the following optimization: Recall that process 0 was called the parent process at level A; however, it would create a serious bottleneck to send all cells to process 0 for testing against line A1. Inspection of the right of Figure 5 shows that *all* processes know the position of line A1. Likewise, for levels B and C of the distribution tree, more than one process knows the position of each line; thus a cell which needs to be tested against a given line can be sent to any of the several processes that know the position of that line. Simple algebraic formulas yield all the processes that know the position of a given line; a process sending a cell can simply multiplex among the possible receivers to avoid bottlenecks.

6 Resampling the Irregular Cells onto the Regular Lattice

The rest of this paper incorporates the above ideas into an implementation of ray casting of irregular scalar data taking the representation of convex polyhedral cells, which draws from [Max 90]. The main contributions of this paper, however, do not depend specifically on the representation that the irregular data takes.

View requests are labelled with a unique view number. A global termination detection algorithm is used to signal that all processes have received all cells that intersect their volume of screen space for a given view.

Upon reception of this signal, each process resamples the irregular data by filling a regular lattice of voxels with scalar values that approximate the irregular scalar values of the cells. Having the values on a regular grid simplifies ray casting.

We iterate the resampling algorithm over the irregular cells. The information contained in each cell includes the position of the center of the cell, the scalar value at the center, the gradient at the center of the cell, and the positions of the vertices. For each cell, it is assumed that the scalar value of the cell is exact at the cell center. A scalar value for all voxels contained within the convex hull formed by the vertices of the cell can be extrapolated using the cell's scalar value and gradient.

If a voxel is entirely contained within one cell, such as voxel 2 in Figure 6, then it is straightforward to extrapolate a scalar value for the voxel. If a voxel intersects more than one cell, such as any of the other voxels in Figure 6, its value can be calculated by one of several methods, in order of expense: first, by giving it the value extrapolated from the cell containing its center; next, giving it the (unweighted) average of values extrapolated from all cells it intersects; or last, giving it the weighted average of values extrapolated from all cells it intersects, based on the percentage of its volume falling into each cell. These last two methods would essentially implement forms of antialiasing.

7 Shading, Color Mapping, and Opacity Mapping

Two alternative shading models are used: the first includes only ambient illumination plus Lambertian reflection. In this model, the intensity of light reflected from a voxel is a function of the intensity of ambient light, the direction and intensity of a light source, and the direction of the gradient of the scalar field. Objects will not cast shadows. A second model with true shadowing is implemented by an operation immediately after loading a new volume; this operation transforms the cells into light-source-space instead of screen-space and then performs shadow casting instead of ray casting.

The shading factor determined by the shading model for each voxel is a floating point number in the range [0,1]. The color map is a user-defined mapping from the scalar value to an RGB color. An RGB color is a set of 3 integers between 0 and 255. The opacity map is a user-defined mapping from the scalar value to an opacity value, which is a floating point number in the range [0,1].

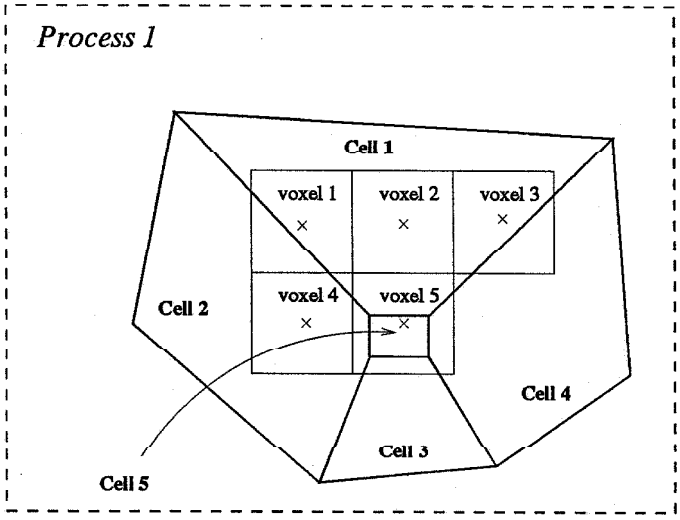


Figure 6: Resampling the Irregular Cells onto the Regular Lattice

To get the final color of a regular lattice point, its scalar value is taken through the color mapping to get a raw RGB color; multiplying the elements of the raw RGB color by the shading factor yields the final RGB color; this will darken the color to an extent appropriate to the amount of shading.

The final RGB values and the opacity values, defined on the regular lattice, are the input given to the ray casting algorithm.

8 Ray Casting

In transforming to screen-space, the Z axis has been transformed to be in the direction of rays from the viewer's eye. When ray (x, y) is cast straight back in the Z direction, it follows the path from the eye of the user, through screen pixel (x, y), and through the regular lattice of data. Rays will iteratively accumulate opacity as they pass through voxels according to this function [Drebbin 88]:

$$\alpha_{x,y,k+1} = \alpha_{x,y,k} + (1 - \alpha_{x,y,k}) * V^{\alpha}_{x,y,k}$$

where $\alpha_{x,y,k}$ is the total opacity accumulated up to iteration k by the ray at screen position (x, y); and $V^{\alpha}_{x,y,k}$ is the opacity of the voxel at position (x, y, k). Color is accumulated in a similar way:

$$\begin{aligned} R_{x,y,k+1} &= R_{x,y,k} + (1 - \alpha_{x,y,k}) * V^{R}_{x,y,k} \\ G_{x,y,k+1} &= G_{x,y,k} + (1 - \alpha_{x,y,k}) * V^{G}_{x,y,k} \\ B_{x,y,k+1} &= B_{x,y,k} + (1 - \alpha_{x,y,k}) * V^{B}_{x,y,k} \end{aligned}$$

where $R_{x,y,k}$, $G_{x,y,k}$, and $B_{x,y,k}$ are the red, green, and blue color values accumulated up to step k by the ray passing through screen position (x, y); and $V^{R}_{x,y,k}$, $V^{G}_{x,y,k}$, and $V^{B}_{x,y,k}$ are the color values of the voxel at position (x, y, k) in the regular lattice.

Ray (x, y) can be terminated early if $\alpha_{x,y,k}$ reaches unity at some k still within the volume. Otherwise, the effect of a background color is added when a ray has passed through the entire volume. When the ray is finished, the pixel at screen position (x, y) is colored by the final RGB values accumulated for ray (x, y).

9 Other Details of Algorithm

9.1 Detecting Completed Loading of all Cells

Each set of irregular cells describing a scalar volume is labelled with a unique volume number. It is known ahead of time how many cells are contained in a given volume. Therefore, to detect termination of the loading (from disk or another concurrently running partition) of a given volume, it is necessary only to iteratively circulate a token from process i to process $(i+1)$ mod n which counts the number of cells loaded so far. When this

number equals the expected number, a "load done" message for that volume is broadcast to all processes.

9.2 Detecting Completed Reception of all Cells

Each view request from the user is labelled with a unique view number. Detecting that all processes are finished receiving all cells needed for a given view is more difficult than detecting completed loading of volumes, because it is not known ahead of time how many cells will be sent. The reception of some cells triggers the sending of shadow copies of those cells; so even when every process has sent all the cells it will send initially, there may be more shadow copies of cells sent later. To detect termination in such a situation, it is necessary to iteratively circulate a token from process i to process $(i + 1) \bmod n$ which maintains the difference between cells sent and cells received. When this token has made the circuit through all processes twice with the value zero, then all sending and receiving is finished [Taylor 89], and a "receive done" signal for that view is broadcast to all processes.

10 Conclusions

This paper contributes several scalable parallel methods which may be applied to the ray casting of irregular scalar volumes.

A scalable dynamic partitioning of the screen-space for parallel ray casting of irregular three-dimensional data is introduced. The dynamic partitioning arranges the data in the parallel machine so that, once the data has been distributed, ray casting from the current viewpoint can be performed without communication between processes. This dynamic partitioning is particularly efficient at rendering a sequence of nearby viewpoints, such as those arising from continuous head motion or right-left stereo pairs of images. For rendering a nearby subsequent viewpoint, most data will not need to be redistributed.

A scalable, memory efficient method using a binary tree to transport the irregular data to the correct processes for ray casting is described. An inexpensive method to load balance a parallel ray casting algorithm partitioned in this way, by dynamically changing the partitioning of the data, is also explained.

A method to implement true shading within this framework by of transforming a newly loaded volume of irregular cells into light-source-space instead of screen-space and then shadow casting instead of ray casting is introduced.

All of these methods are optimized for fine grain multicomputers and low-latency communication and are highly scalable.

11 Acknowledgements

We would like to thank David Ellsworth and Ulrich Neumann of UNC for their helpful advice.

References

[Blinn 82] Blinn, James F. "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces", *Computer Graphics* 16(3):21-29, July 1982.

[Dally 86] Dally, William. *A VLSI architecture for concurrent data structures* Ph.D. thesis, Department of Computer Science, California Institute of Technology, Technical report 5209:TR:86,1986.

[Drebbin 88] Drebbin, Robert A; Carpenter, Loren; and Hanrahan, Pat. "Volume Rendering", *Computer Graphics*, 22(4):65-74, August, 1988.

[Gallagher 89] Gallagher, Richard S. and Nagtegaal, Joop C. "An Efficient 3-D Visualization Technique for Finite Element Models and Other Coarse Volumes", *Computer Graphics*, 23(3):185-194, July 1989.

[Kajiya 84] Kajiya, James T. and Von Herzen, Brian P. "Ray Tracing Volume Densities", *Computer Graphics*, 18(3), July 1984.

[Lorensen 87] Lorensen, William W. and Cline, Harvey E. "Marching Cubes: A high resolution 3D surface construction algorithm", *Computer Graphics*, 21(4):163-169, July 1987.

[Max 86] Max, Nelson. "Light Diffusion through Clouds and Haze", *Computer Vision, Graphics, and Image Processing*, 33:280-292, 1986.

[Max 90] Max, Nelson; Hanrahan, Pat; and Crawfis, Roger. "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions", *Computer Graphics*, 24(5):27-33, November, 1990.

[Sabella 88] Sabella, Paolo. "A Rendering Algorithm for Visualizing 3D Scalar Fields", *Computer Graphics*, 22(4):51-58, August 1988.

[Seitz 91] Seitz, C. L., Multicomputers, *Developments in Concurrency and Communication*, C.A.R. Hoare (ed.), Addison-Wesley, 1991.

[Taylor 89] Taylor, Stephen. *Parallel Logic Programming Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Upson 88] Upson, Craig and Keeler, Michael. "V-BUFFER: Visible Volume Rendering" *Computer Graphics*, 22(4):59-64, August 1988.